

Reef
Ph.D. Thesis Proposal

Piotr Kaminski
July 5, 2004

Table of Contents

Table of Contents	1
1. Introduction	2
1.1. Documentation	3
1.2. Unified Modeling Language	5
1.3. Costs and Benefits	6
2. Hypotheses.....	7
2.1. Wonders of UML (H1-H3).....	7
2.2. Travails of UML (H4-H6).....	10
2.3. Idiosyncrasies of Software Engineering (H7-H9).....	12
3. Tool Specification	15
3.1. Requirements.....	15
3.1.1. Primary Use Cases (U1-U2).....	16
3.1.2. Secondary Use Cases (U3-U6).....	18
3.2. Architecture	19
3.3. Back-end Design.....	20
3.3.1. Data Model.....	20
3.3.2. Data Flow	22
3.3.3. Platform.....	24
3.3.4. Features	25
3.4. Front-end Design	28
3.4.1. Platform	28
3.4.2. Data Management and Communication	32
3.4.3. User Interface.....	33
4. Conclusions.....	38
4.1. Research Plan.....	38
4.2. Expected Contributions.....	38
Bibliography	40
Appendix A. Sample Edit Action List.....	48

1. Introduction

Software evolution is a fact of (modern) life. As computers inveigle themselves into every aspect of human existence more and more software comes into daily use. Without fail, this software needs to be fixed, extended or adapted to changing circumstances—and despite our best efforts at minimizing dependencies, the modifications have an unfortunate tendency to snowball. Rebuilding the software from scratch is often not a viable option due to the high risks and costs involved, leaving gradual evolution as the only realistic approach.

However, software evolution brings its own set of issues to the table. Successfully modifying any construct requires at least a partial understanding of it [Sta84], and that understanding can be difficult to gain [Cor89]. Even if the attempt goes swimmingly, the changes usually increase the complexity of the software, making the next effort commensurately more difficult. There is some truth to the claim that, untended, software systems gravitate towards incomprehensibility.

Why is software so difficult to understand? Some claim that it is the most complex artefact ever designed by mankind: the human mind simply cannot keep track of the myriad details contained in the source code, failing to see the forest for the trees. The proven solution is to tame the complexity by raising the level of abstraction at which the developer perceives the majority of the system, concentrating on only a manageable quantity of details at any given time. During initial development of a system, these abstractions drive the implementation and are thus naturally grasped by the developers. However, when the project transitions into its maintenance phase, knowledge of the abstractions quickly dissipates¹ due to the diminished pace and personnel rotation. This *loss of abstraction* is in large part what makes software evolution so difficult.

¹ This is true even for projects that stay in “active” development on a continual basis, only the scale changes: individual subsystems enter the maintenance phase as active development moves on.

1.1. Documentation

Since source code drowns developers in details, the traditional, rational response is to somehow provide additional information about the software at a higher level of abstraction. This documentation varies in form, author and time of creation; it is not clear which kinds of documentation (if any) are beneficial to software evolution in various circumstances (see Section 2.1). The following subsections discuss some common variations on this theme, emphasizing their deficiencies.

Source Code Comments | The easiest and most common way to record abstractions is in source code comments. This practice is traditionally taught in introductory software engineering courses and has seen widespread adoption in varying degrees. Comments are convenient for a developer to write (no need to switch documents) and can be read either as part of the source code or extracted into separate documents (Javadoc (<http://java.sun.com/j2se/javadoc/>) and Doxygen (<http://www.doxygen.org/>) being the most successful examples of the latter practice).

Even though the costs are low, the immediate benefits to the developer are low as well, so comments do not always get written or kept up-to-date. Source code comments are also limited to short, localized pieces of text, whereas certain abstractions are best expressed as diagrams or longer, coherent sections of prose. Comments are thus necessary but not sufficient for documenting software.

Design Documents | The development of most software systems, especially large ones, usually starts with some high-level documentation in the form of requirements, use cases, and architectural decisions. Through analysis and design iterations, these documents then get progressively refined into running code. There is a strong temptation to retain all of these design documents to assist with later evolution, preserving traceability all the way into the code; in fact, many popular methodologies espouse this approach.

Unfortunately, this approach leads to an inflexible process that is hard-pressed to respond to changing requirements and the developers' understanding of the domain. The

initial design documents are often purposely incomplete, informal or vague, meant to be used as a guiding sketch then discarded [Fow04 p. 2]. If they are to be preserved, they must be formalized and all modifications to the source code must be separately documented, often at multiple levels of abstraction. The extra work has no immediate benefits for the developer, who has a deadline to meet and is so immersed in the concrete software that the abstract changes are obvious. Consequently, design documentation is rarely kept in sync with the code.

There are two distinct responses to this problem. One is to acknowledge the failings of formal documentation and leave developers free to update or discard documentation at will, deemphasizing traceability. This approach is exemplified by the Agile Modeling methodology. The other response is to formalize the development process, emphasizing strict models over code and imposing discipline through round-trip engineering tools. The emerging Model Driven Architecture (MDA) movement seems to epitomize these values. Neither response is very satisfying: the former often leaves a software system with no useful documentation, while the latter straightjackets the developers, reducing their productivity.

Reverse-Engineered Diagrams | If software documentation was not written during forward engineering, all is not lost: it is possible to reverse-engineer some of the system's abstract principles from the code. The process can be somewhat automated, with specialized tools (e.g. Rigi [Won98], others) extracting high-level features and inferring certain patterns. The end result is usually a set of diagrams (often graphs) that more-or-less represent the structure and behaviour of the subject system and can serve as a guide to the code.

However, this process is necessarily imperfect [KS+02] as the implementation of an abstract model is not an exactly reversible transformation [GA03]. Some abstract features are diffused beyond recognition or disappear altogether, while other irrelevant “phantom” properties emerge spontaneously from the code. These flawed results are exacerbated by primitive automated layout facilities [Eic02b, EG03] that, for even moderately

sized systems, produce indecipherable renditions such as that displayed in Figure 1. Research into improved algorithms is ongoing (see Section 3.3.4), but at the moment reverse engineered diagrams need a disheartening amount of human attention to look presentable.

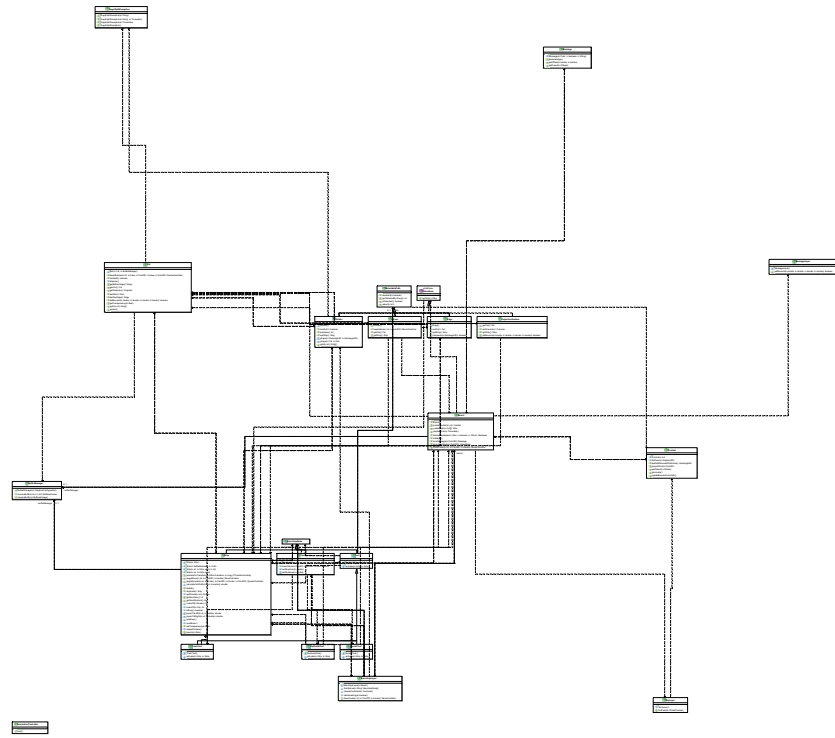


Figure 1. A diagram of 23 classes after automatic layout

Whole-system reverse-engineering is indubitably a painful exercise and a distant second to having access to ready-made documentation. The questions, then, are (i) what form should the extended documentation take and (ii) how to maximize the chances that it will be produced?

1.2. *Unified Modeling Language*

To maximize the usefulness of the extended documentation we need to find a good trade-off between its format's expressiveness, density, and familiarity to developers. For example, commented source code is fairly expressive (since it can represent most abstractions) and very familiar to developers, but its dispersed nature makes it difficult to form high-level pictures. Generic labelled directed graphs can be made to express

most anything, but the simple notation has low density and the lack of standards prevents easy interpretation by developers. Logic systems, such as Pi calculus or F-logic [KLW95], are dense and highly expressive but completely incomprehensible to most developers.

There is no universally optimal solution, so for this project I will constrain my inquiries to object-oriented systems. In this domain, the Unified Modeling Language (UML) [Fow04, OMG03] is the clear choice for abstract documentation. It is quite expressive (especially when supplemented with the Object Constraint Language (OCL), though at the expense of familiarity), and its graph-based representation is reasonably dense. Its core constructs are well-defined and familiar to object-oriented developers who have even a passing acquaintance of industry trends over the last few years.² It is also sufficiently flexible to model perspectives ranging from analysis to design to implementation, letting the writer fine-tune the documentation's level of abstraction.

Though concentrating on UML restricts the potential audience, many of the techniques introduced in Section 3 could be applied to other graph-based representations as well.

1.3. *Costs and Benefits*

As mentioned above, few developers bother to produce UML diagrams that describe their systems; the reasons behind these inactions boil down to a perception that the costs are too high and the benefits too few [Zei02]. Hence, to increase adoption of UML documentation in the development process, we must reduce the costs and expand the benefits, or at least improve the developers' perceptions of these aspects.

The costs are the usual culprits, adversaries of adoption everywhere: complexity, expense, lack of support or polish, bad integration into an existing workflow [BJ+03]. Many of these concerns can be addressed with a carefully designed and well-engineered adoption-centric tool; Section 3 presents a concrete proposal for just such a tool. Sections 2.2 and 2.3 put forward a few hypotheses that form the theoretical foun-

² Surveys indicate that UML's penetration is low (around 34% in June 2002 [Zei02]), but it is my contention that many more developers can read UML than choose to write it.

dation of the proposed design and the answers to which might explain why the current crop of tools have failed to take the software development world by storm.

Much has been asserted about the benefits of UML diagrams with surprisingly little validation. Section 2.1 restates some commonly accepted hypotheses and reports on related research. As a further deficiency, most of the supposed benefits are long-term and reward third parties—hardly a potent motivation for the developer who needs results right *now*. To remedy this shortcoming, Section 2.1 posits a few ways in which keeping UML diagrams up to date might also help the developer.

2. Hypotheses

This section lays out a few hypotheses related to UML diagrams used as documentation and software engineering practices. Many of the hypotheses are likely to be quite uncontroversial, but are stated for completeness' sake. A longer discussion of the reasons and evidence for and against each hypothesis follows its statement.

To avoid excessively nebulous discussions, I make use of the following more easily measurable variables in the hypotheses:

- *speed* to mean a reduction in the time needed to complete a task;
- *accuracy* to mean a reduction in the number of errors in a task's result;
- *quality* to mean an increase in the design quality of a product; and
- *performance* to mean any combination of the three.

The variables are probably not independent and quantifying any correlation between them could be interesting as well.

2.1. *Wonders of UML (H1-H3)*

The hypotheses in this section explore various aspects of the claim that UML diagrams contribute to software understanding.

(H1) Documentation in the form of UML diagrams that are automatically updated during development and reviewed by the developer increases the developer's accuracy and quality, and increases the development team's performance.

This hypothesized benefit is the most important, since it directly impacts the developers' work. Though reviewing the diagrams takes time (and hence will not increase the developer's speed), it can help spot high-level bugs (e.g., introducing an undesirable dependency) and keep a handle on the quality of the design (see (H2)). For the rest of the team, regularly updated UML diagrams make integration easier, and are superior to raw source code deltas for tracking changes. Furthermore, the team lead can keep a handle on architectural drift and nimbly steer the project away from danger before the code has had a chance to set.

(H1) is in good company, with other projects trying to increase developers' productivity by closing a feedback loop. For example, JUnit automates regression testing, giving the developer a nearly instantaneous red signal when the code fails a test. Hackystat [Joh03] automates the collection of certain metrics in an attempt to keep a development team informed about their project's progress. There are many other efforts in a similar vein, but I have not been able to find much formal discussion or empirical measurements of their effectiveness.

(H2) Documentation in the form of UML diagrams allows for fast evaluation of the quality of a system's design without forming an understanding of the system.

Though UML diagrams can help with system understanding, this hypothesis claims that the diagrams' shapes themselves are closely correlated with the design's quality. In other words, the analyst does not need to parse and integrate the details of the semantics conveyed by the diagrams, but merely glance at their topography, relying on the superior human pattern-recognition skills. Naturally, this presupposes that the diagrams are nicely laid out (see (H6)); no matter how good the design, it is always possible to draw a ghastly diagram.

This hypothesis is backed up by my personal experience marking projects in the Software Engineering 330 course; within minutes of looking at the diagrams provided by the students I formed an initial impression of the design's quality (and its author's competence, which is closely related) that was usually born out by further detailed examinations of the source code.³ More formally, there has been an initial attempt to relate the shape of class diagrams to object-oriented metrics [Eic03], but with no empirical evidence thus far.

Should these pioneering investigations pan out, the postulate might be extended to correlate specific design principles (e.g., indirection, cohesion) to visual patterns.

(H3) Documentation in the form of UML diagrams improves the performance of third parties in integration and maintenance tasks.

It is fairly well accepted that an understanding of (the relevant parts of) the software is critical to performance on maintenance and integration tasks [MV95], and that documentation increases the speed, accuracy and quality of understanding [Vis97]. Moreover, one experiment indicated that the performance advantages conferred by superior software development skills are voided in the absence of documentation [Try97], further increasing the importance of documentation to organizations trying to get their money's worth from (expensive) highly skilled employees.

The jury is still out on whether UML is an effective form of graphical documentation [TH03] [PC+01] [PC+02], but surely its popularity in the industry must stem from some noticeable benefits rather than just being the result of the Object Management Group's advocacy efforts. I conjecture that UML documentation will have an overall positive impact on maintenance performance, with the greatest improvement for adaptive maintenance, smaller for preventive and perfective maintenance, and smallest for corrective maintenance. The rationale is that adaptive maintenance tasks require the most abstract understanding—the province of UML diagrams—while corrective maintenance tasks

³ It is eminently possible that my final opinion was swayed by my initial impressions, so a proper experiment would need an appropriate blinding protocol.

require detailed understanding that can only come from the source code, obviating the need for design diagrams.

2.2. *Travails of UML (H4-H6)*

This section concentrates on the obstacles to the production and understanding of UML documentation.

- (H4) Documentation in the form of UML diagrams updated throughout the development process is of superior accuracy to UML diagrams produced before development has begun or after development has ended. It takes longer to keep UML diagrams up to date during development than to produce them all at the same time. However, the time spent is perceived to be shorter by the developer in the former case.

There seems to be a wide variety of opinions on when the design of a software system ought to be documented. Traditional waterfall processes prescribe that the system be designed and documented up front. Though still in use [NL03], waterfall methodologies have generally been discredited for most types of software projects as they have proven too brittle. It is unlikely that the initial design will survive the coding phase, yet the process makes no allowances for feeding changes back up the waterfall, so the documentation is doomed to be incorrect.⁴ The converse approach of documenting after the fact has the advantage that complete information about the system is available, but the developers have already forgotten many of the design's important details. Post-facto documentation tends to be superficial and rushed.

Documentation in hindsight has the benefit of a working system and weeks/months of experience. Documentation in foresight is documentation based upon conjecture. Neither is typically any good. [Bla00]

⁴ The Model Driven Architecture (MDA) movement takes a stab at this problem by prescribing a completely automated transformation from design models to code; by definition, MDA models are always accurate. The merits of the MDA approach are a matter of some debate, but I believe its success will be limited to a mostly irrelevant subset of well-understood waterfall-friendly projects [Amb03b].

Incremental refinement should lead to more accurate documentation, but I have not found any empirical studies to back up this hypothesis—perhaps the conclusion appears too obvious. On the other hand, incremental methods often decrease the speed of development, which could make them a hard sell. We might be able to conceal this shortcoming by making the iterations as short as possible and increasing their number (e.g., 15 minutes every day). The total time spent would be the same (or even longer), but perhaps the developers would perceive the smaller tasks as less onerous.

As you can surmise from the above, this hypothesis is very tentative and requires more research into psychological factors and a solid empirical study to determine its truth.

(H5) There is an optimal window of opportunity for a developer to update documentation to match changes to the source code. This window extends for approximately 24 hours from the time the code is committed.

If we want accurate documentation and we assume that code comes first (see (H8)), the documentation must eventually be updated to match changes to the source code. As mentioned in (H5), the update must not come too late or the developer risks forgetting important details that would make the documentation inaccurate or incomplete. Conversely, the update must not be contemporaneous with code development as it distracts the developer from the task at hand, reducing productivity and increasing perceived documentation cost. Also, during development the developer is well aware of the changing structure of the code, so he gains no benefit from an updated abstract model.

It follows that there is an optimal window of opportunity for the documentation update. The value of 24 hours is an initial guess based on personal experience, but it is likely to vary depending on the circumstances and would need to be refined via experiments.

(H6) UML diagrams that are nicely presented increase the performance of tasks that require program understanding.

It is well known that visual structure affects memory [Kem99] and understanding [Tuf97], and experiments have confirmed that the same factors affect UML [TH03,

PAC00] and other graph-based model visualizations [HLN04]. Of course, opinions differ on what makes a nice presentation. There are various high-level guides for software engineers who draw UML diagrams [Amb03a, MM03], recommendations targeted at specific notational variations [PC+02, PC+01], and more-or-less computable aesthetic criteria employed by automatic layout algorithms [EKS03a, Eic02a, KG02].

While more experimental results are always welcome, it seems safe to accept this hypothesis as proven. The aesthetic criteria are more contentious and difficult to isolate but it would once again seem safe to select a common subset, keeping in mind that it is not possible to please everyone simultaneously.

2.3. *Idiosyncrasies of Software Engineering (H7-H9)*

This section lists three hypotheses that concern the adoption and use of tools for software engineering activities. Unfortunately, the predictions concern effects that are hard to quantify, so verifying these theories may prove difficult to the point that they should perhaps be treated as axioms.

(H7) A tool's adoptability is increased by its benefit to the user and decreased by the magnitude of required changes to the user's workflow.

It should come as no surprise that the more useful the tool, the more likely it is to be adopted. However, unless a tool offers truly ground-breaking benefits (e.g., email, the web), its adoptability will be moderated by how well it fits into a user's existing workflow. For example, a tool might be adopted if it has limited benefits but could be dropped right into an existing process (e.g., a minor update of a tool already in use), whereas it would be ignored if it required a change in procedures. Naturally, there are many other factors affecting adoption [BJ+03], but these two seem to be the most relevant to the tool proposed herein.

How does this hypothesis relate to software documentation? Considering Section 2.1, and barring external social or economic constraints that can have deleterious effects on morale ("Document or you're fired!"), producing documentation often brings little di-

rect benefit to a developer. According to this hypothesis, any documentation tool must therefore fit very well indeed into a developer's workflow if it is to stand a chance of being adopted.

(H8) The ground truth of a software system is its source code.

The development of a software system usually produces a wide assortment of artefacts, from requirement lists and analyses to bug reports and code comments. It is rare that all of them agree, either due to errors or simply because they did not keep up with the system's evolution. In these situations, though various documents may indicate what the system *was* or *should be*, the source code⁵ provides the ultimate measure of what the system *is*. The accuracy of all other artefacts must be judged against the reality of the code.

A corollary is that source code is highly prized by developers and a tool's automated code generation or mutation must strive to be transparent in purpose and minimally invasive.

(H9) Current approaches to round-trip and "tripless" integration between source code and UML diagrams are fatally flawed.

Based on (H8), it is clearly important that UML diagrams be synchronized with the source code. This is difficult to achieve with a typical stand-alone diagram editor: changes to the diagram may not be correctly implemented in the code, while ad-hoc code modifications are not reflected by the diagram. In response to this problem, many tools offer round-trip engineering facilities that emit skeletal code based on the diagrams and can reverse-engineer updated source code back into a model. Unfortunately, the code skeletons are so simple as to not be worth generating and the unsophisticated reverse-engineering algorithms fail to extract a good portion of even the recoverable

⁵ By "source code" I mean all digital resources that are transformed into an executable system by the build process. For this project, I am not interested in legacy systems whose source code cannot be rebuilt. While MDA-like systems do technically fall under this definition (the models are the "source code" according to my definition), I am not interested in these kinds of system either.

subset of design features. Most damning is that the tools usually fail to support an iterative development process, reverse-engineering the diagrams from scratch every time. Tripless tools—a modern take on roundtrip engineering represented in tools such as Together (<http://www.borland.com/together/>) and EclipseUML (<http://www.omondo.com/>)—tightly couple diagram editors and integrated development environments (IDEs). In these environments, the UML diagrams and the code are but two representations of a single underlying model, and editing one also modifies the other.⁶ This solves the synchronization problem, but raises serious new issues of its own.

First and foremost, the diagrams thus produced reflect the source code in every minute detail. Far from being an advantage, this discards the greatest benefits of modeling: the superior expressivity of UML and its power of abstraction. UML has constructs that preserve developers' intent, intent that is often lost when the model is translated to code. For example, UML's association classes, composition associations, constraints, and certain multiplicities encode important properties of the model but have no equivalents in most programming languages. Furthermore, to be useful, UML diagrams must selectively elide excessive clutter, raising the model's level of abstraction to help the viewers' minds grasp larger pieces of the whole [Bel04b]. The diagram is still drawn from a software perspective [Fow04 p. 5], but at a design rather than implementation level.

Implementation-level tripless diagrams are normally used for two functions: code navigation and refactoring [Fow00]. However, both are better delivered directly at the code level. An IDE's outlining and linking services provide superior context awareness, and its refactoring tools are syntax-aware and thus less invasive (see (H8)). Thus, implementation-level diagrams fail to enhance understanding without improving on facilities already provided by a typical IDE.

⁶ Not all of UML's 13 kinds of diagrams can share a model with code. Typically, class, package and sometimes interaction diagrams are supported; other diagrams are either not available or not linked to the code.

It would be possible to make the diagrams more abstract and expressive while retaining a connection to the code—for example by a tripless version of the tool described in Section 3—but there would still be little point in hosting such a tool within the IDE. It is my experience that the activities of modeling and programming are mutually exclusive. When modeling, I do not want to worry about the code that is being generated or mangled; similarly, when coding I will usually keep the model in mind (or in view, even), but do not want to be bothered with decisions about which features to abstract, how to lay out new elements, etc. While I may switch between the two activities often throughout a day, they are best kept separate, and any automatically propagated changes clearly indicated for my review when I next shift. A tripless modeling tool could certainly respect these constraints, but would gain little from its integration into the IDE.

At present, many people seem enamoured with the idea of round-trip engineering [CTM03], though there are a few dissenting voices in the wilderness [Hol02]. The opinion of software developers is unknown; since all top UML modeling tools include either round-trip or tripless engineering as one of their features, a measure of the tools' popularities would not be indicative of the developers' desires in this matter. There is some anecdotal evidence that the tools' code generation facilities are very rarely used, though [Sho04].

3. Tool Specification

This section specifies the requirements and an initial high-level design for a UML diagramming tool. The requirements and design were driven by the hypotheses presented in Section 2, and the tool should in turn provide a platform for verifying some of those propositions.

3.1. Requirements

The overall goal of the tool is to help developers efficiently create and maintain UML diagrams that effectively impart an improved understanding of the system to their

readers. It is not a goal to have the tool compete with scrap paper and whiteboards for up-front analysis and design activities.

The tool's adoptability is a priority in all requirements.

Since the Reef tool is software that operates on software, there is potential for confusion when describing activity flows. For clarity, I always use "tool" to mean the Reef tool, and "system" to mean the system being developed and documented.⁷

3.1.1. Primary Use Cases (U1-U2)

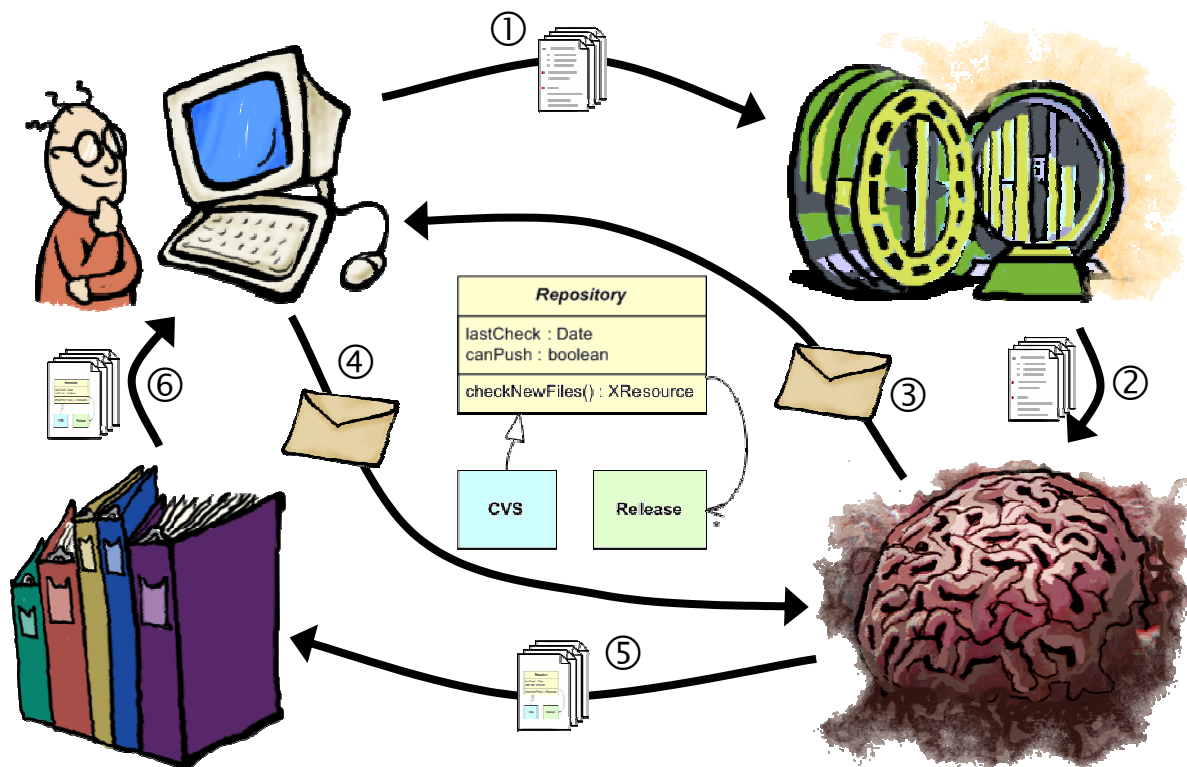


Figure 2. Primary scenario sketch

(U1) Update UML diagrams after the system changes.

The developer modifies the system's source code and ① commits the newest version into the code repository. The tool ② detects the event, parses the changed source code,

⁷ Of course, I expect that during development "tool" and "system" will be the same, as Reef is used to document itself. Eat your own dog food and all that.

compiles the project and runs instrumented unit tests. The tool then reverse-engineers both static and dynamic implementation-level models of the system and updates the system's UML diagrams (creating new ones if necessary). If the developer specified any "standing orders" when editing diagrams in the past (e.g., "don't show private inner classes"), the tool adjusts the diagrams accordingly. The tool ③ emails diagrams that have been significantly modified to the developer, with the changes highlighted.

The developer reviews the diagrams as time permits, perhaps comparing them to the initial design sketches. If necessary, the developer edits the (changes to the) diagrams to raise the level of abstraction, capture design rationales, and re-introduce design features that became unrecognizable in the translation to code. Based on the edits, the developer also sets standing orders to automatically apply changes to current and future diagrams according to simple rules. When the developer is happy with the diagram, ④ he approves it and sends it back to the tool. If the developer is unable or unwilling to bring the diagram to a satisfactory state, he can delegate it to somebody else, split it into smaller diagrams, or tell the tool to discard it altogether.

The tool ⑤ integrates approved diagrams into the system's documentation (e.g., Javadocs) and processes any new standing orders, then notifies interested parties that the diagrams have been updated.

(U2) Use UML diagrams to help system understanding.

A developer—not necessarily the system's original designer or implementer—needs to gain an understanding of the system. He browses through the system's documentation, ⑥ which includes diagrams that clearly indicate the last time they were validated. He can navigate between diagrams by following hyperlinks, in both the documentation text and in the diagrams themselves. He can adjust the diagrams' display characteristics, and even opportunistically correct and update them if authorized to do so. The tool collects statistics on the relative popularity of the diagrams to help the developers in (U1) and (U4) decide whether to invest the time to update the diagram or just throw it out.

3.1.2. Secondary Use Cases (U3-U6)

(U3) Use UML diagrams to help evolve the system's design.

The developer wishes to change the design of the system. He locates the relevant diagrams in the system's on-line documentation and edits them to reflect the desired form of the system. When the design is done, the tool collects all the edited diagrams and highlights the changes, to make it easier to see what needs to be implemented. When the modified code is committed, the developer can compare the reverse-engineered diagrams against the ideal ones and resolve any differences before approving the lot.

(U4) Manage a project's diagrams.

A team leader or manager wants to check the status of the project's design diagrams. The tool provides reports on stale diagrams, diagram update and consultation frequency, developers' diagram editing efforts, etc. Based on the information presented, the manager can forward stale diagrams for revision to selected team members, delete unimportant diagrams, etc. The manager can also decide to expose a summary of the most important metrics on a "project dashboard", to keep the team up to date about the state of the diagram documentation.

(U5) Configure the tool for a project.

The developer wants to start documenting a system using the tool. The developer inputs the code repository's connection parameters and his email address. If the system has source code, the tool checks it out and proceeds to create and send out new diagrams as in (U1), as if though all the code had just been committed into an empty repository.

Other configuration options could include setting the means of communication (email, instant messaging, RSS) and associating multiple projects to share standing orders. However, the developer must be able to initially set up the tool with a minimum of effort.

(U6) Customize or extend the tool for different purposes.

The developer wants to adapt the tool to his needs, either by replacing existing components or by adding new components to the framework, leveraging the existing functionality and data model. To encourage a vibrant plug-in scene, the internal data structure and process flows of the tool should be easy to understand, and plug-ins should be able to share data while not interfering with each other's operation by default.

3.2. Architecture

This section provides a high-level overview of the tool's design, as shown in Figure 3.

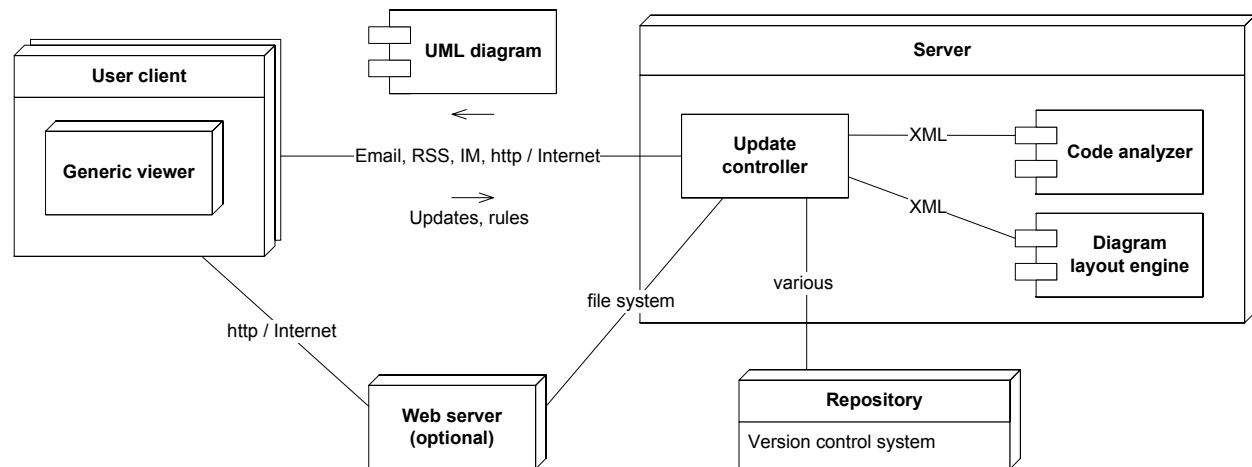


Figure 3. Tool architecture and deployment diagram

To maximize adoptability, the tool is designed to be minimally invasive. The users will not need to install the tool on their machines; at most, they will be required to set up a viewer that is not specific to this application (e.g., a virtual machine of some kind). Hopefully, the viewer will be popular enough (see Section 3.3.3) that it will already be available on most machines, easing the way for a viral⁸ spread of the tool. This deployment strategy also lets the back-end engage in computationally intensive tasks without engendering a perception that the tool is slow, unlike a desktop-bound application.

⁸ I mean viral in the benign sense of “viral marketing”.

For these reasons, all custom software is installed on the back-end web server, while the front-end of the tool is transmitted to users as part of the diagrams. The primary means of communication is email, as it is ubiquitous, push-oriented (receiving email requires no explicit action on the user's part), typically stored or cached locally and accessible off-line (e.g., during a long-haul flight), and integrated into the user's task management processes. Other communication methods can also be deployed to increase the chances of the tool fitting into the users' workflow: instant messaging (IM) reduces update latency, RSS⁹ allows for multiple anonymous receivers, and web access puts the control back in the users' hands by being a pull service.

The following subsections explore the design spaces for the tool's back-end and front-end in more detail.

3.3. *Back-end Design*

This section delves into the design of Reef's back-end, starting with the underpinnings of the data model, through an overview of the functional and data flows, and finally with details of some of the more interesting components.

3.3.1. Data Model

All of the back-end's responsibilities revolve around extracting and manipulating information about the system, so the data model and storage are critical cross-cutting concerns for the tool. The model should be semi-structured [ASB99] to permit an exploratory approach to development unfettered by onerous schema alterations, and to eventually allow multiple independent extensions to the tool to cohabitate without tricky schema integration. The model's syntax should also be easily readable in its native format to simplify debugging and increase adoption thanks to the well-known "view source" effect [Shi98]. Finally, the model must have free database implementations available to minimize the impedance mismatch and ultimately ensure its scalability to large systems.

⁹ The acronym "RSS" expands to "Rich Site Summary", "RDF Site Summary" or "Really Simple Syndication", depending on who you ask—the acronym is about the only thing all the parties can agree on. An upcoming remake of the standard may be called "Atom" (which does not expand to anything).

I have quickly evaluated a number of models according to the criteria above; the results are summarized in the adjacent table. The classic relational model has the advantage of decades of development, and its strict

Model	Criterion		
	<i>Flexible</i>	<i>Accessible</i>	<i>Supported</i>
<i>Relational</i>	somewhat	somewhat	yes
<i>MOF</i>	no	no	somewhat
<i>RDF</i>	yes	somewhat	somewhat
<i>Braque</i>	yes	no	no
<i>GXL</i>	yes	no	no
<i>XML</i>	yes	yes	yes

schema requirements can be partially overcome with careful use of a multitude of keyed tables (as done in softChange [Ger04]). Even so, the scattered normalized data tables make queries unintuitive and require a lot of up-front planning, making relational databases inappropriate for this exploratory project. The Meta-Object Facility (MOF) [OMG02a], the model behind UML, is just as strict and generally considered unapproachable by developers; XML Metadata Interchange (XMI) [OMG02b], the MOF's serialization format, never really took off. The Resource Description Format (RDF) [RC04], on the other hand, was designed purposely for knowledge federation, but its primitive triples substrate makes serializations difficult to comprehend and the tools supporting it lack maturity. Playing in the same design space, Braque [Kam02a] has a more sophisticated model, but cannot be meaningfully serialized and has barebones tooling. The rich Graph Exchange Language (GXL) [Win01], agreed upon by the software engineering community for exchanging models of software, was neither meant to be human-readable nor intended to be used as a database model, so it fails to satisfy this project's selection criteria.

Surprisingly, the Extensible Markup Language (XML) [BPS00] proves to be an outstanding candidate. Used without schemas it allows complete flexibility while preventing collisions thanks to namespaces [BHL99]. Moreover, it offers ordered hierarchical containment—a natural way to model source code—as a model primitive, a feature unmatched by any of the others save Braque. Its syntax reflects the model directly and is well-known by developers, and a fair amount of XML databases are available, both open-source and commercially [Bou04]. XML's main drawback is the model's lack of support for non-hierarchical relationships, but this is partially palliated by the advanced

XPath [BB+03] and XQuery [BC+03] query languages implemented and optimized by the databases.

I have provisionally chosen XML as Reef’s central data model language. Since XML databases are a relative newcomer to the data management scene, an evaluation of their merits (or lack thereof [Pas04]) should prove valuable in and of itself as well.

3.3.2. Data Flow

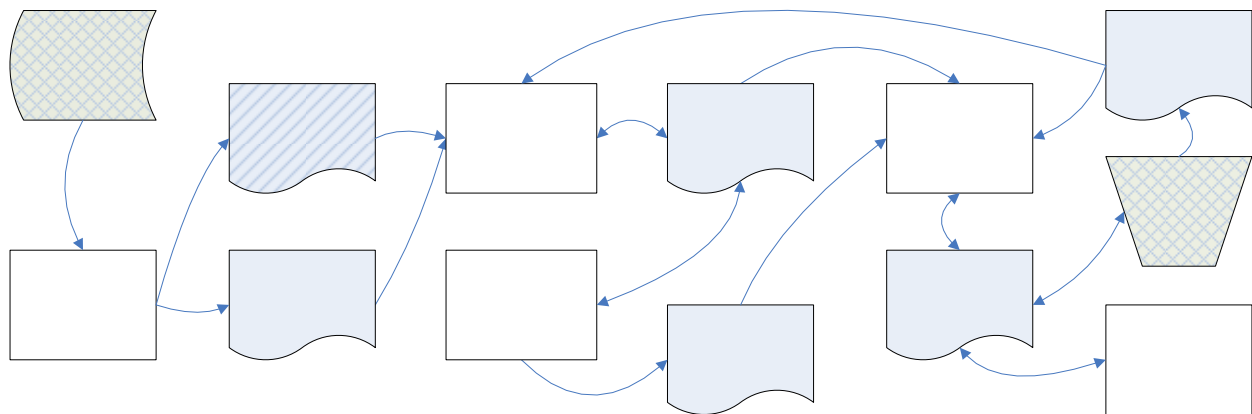



Figure 4. Back-end data flow diagram

Figure 4 shows an overview of Reef’s proposed data flow. Source code is extracted from the repository with an adapter, which checks out and (optionally) builds the code and generates a list of files changed since the last run to cut down on unnecessary parsing. A series of fact extractors parses the relevant code files, updating a language-specific code model. These extractors may include a source code extractor, object code extractor, a dynamic execution trace extractor, etc. The model is further filled in by fact processors (e.g., a type resolver, aspect applicator, etc.), and finally an identity fuser correlates new model entities with old ones, producing a list of model changes at a fine granularity. All the fact extractors and processors are language-specific.

Next in the pipeline is a language-specific diagram extractor, which updates language-neutral diagrams based on the changes to the code model. Any modified diagrams are then incrementally laid out by a language-neutral diagram layout engine that makes an effort to preserve the layout of unmodified elements. The diagrams are then ready to be

further edited by the user, who can selectively amplify any action he takes (see Section 3.4.3). These amplifications are aggregated in a rule base that is used in later runs of the extractors. The diagram extractors use the rules to decide whether and how to introduce elements into the diagrams. The fact extractors may use the rules in more creative ways to drive fact extraction. For example, if the user specified that a class is a collection and should be drawn as an association, and the static fact extractors have failed to ascertain the type of the collection's contents, the dynamic extractor may choose to instrument the code specifically to gain this information.

Note that there is no intermediate abstract domain model: the implementation-level code model is transformed directly into diagrams, and each code entity may well generate multiple diagram elements. This differs from most “professional” UML tools that insist on maintaining an independent abstract model that is then viewed through the diagrams. When working with those tools, the user must remain aware at all times whether he is making modifications to the underlying model or merely the view. For example, deleting a class has one of two meanings: deleting its projection from a diagram, or deleting the entity from the model and consequently its projection from all diagrams. It seems to me that maintaining a separate abstract model is an unnecessary complication that produces no value for the user, who is only interested in the diagrams,¹⁰ and action amplification will prove a more intuitive way of effecting system-wide changes.

Note also that nearly all data items in this flow are XML (represented by ) , so it should be easy to insert additional extractors or processors. It should even be possible to insert matching pairs of fact and diagram extractors that communicate custom information through the code model without upsetting any other components, thanks to the transparent extensibility of XML documents when queried properly.

¹⁰ This assumes that the UML diagrams are used for informal communication. In formalized code generation processes (such as the MDA) the abstract model is clearly paramount, but as mentioned in footnote 5 the Reef project does not cater to these methodologies.

3.3.3. Platform

There still remains the practical question of which platform to use to support the architecture sketched out above. I have chosen to program the back-end in Java, since I know the language well, it is well suited to back-end development, and its popularity ensures a high level of third party support. As part of my preliminary investigation, I have located several open-source libraries that would speed the development of Reef:

- *eXist* (<http://exist-db.org/>), an XML database written in pure Java that provides document storage with automatic structural and full text indexing and highly optimized collection-wide XPath and XQuery querying.
- *CruiseControl* (<http://cruisecontrol.sourceforge.net/>) and *Anthill* (<http://www.urbancode.com/projects/anthill/>), two continuous integration applications that provide a number of repository adapters, automatic builds and various notification options. CruiseControl is especially interesting as it has a well-developed plugin interface and uses XML to communicate data between modules.
- *QDox* (<http://qdox.codehaus.org/>), a fast Java superstructure parser and *ASM* (<http://asm.objectweb.org/>), a fast bytecode parser. I initially plan to implement Reef for Java, since it is popular yet easy to parse. Adding parsers for other languages (e.g., C# or ECMA Script) would improve Reef's appeal and allow investigations into multi-language projects, but is not critical to the proposed dissertation.

Finally, while Java is a robust language, its static typing and lack of advanced features make programming a notoriously high-ceremony affair. Dynamic scripting languages like Python and Ruby claim to improve productivity by stripping away much of the “noise” and allowing developers to create elegant new constructs to make the code resemble a Domain Specific Language (DSL). It might be worth investigating these claims in Reef: Figure 5 contrasts a Java code fragment to its equivalent written in a slightly extended version of Groovy (<http://groovy.codehaus.org/>), a new scripting language that can integrate tightly with Java.

<i>Java code fragment</i>	<i>Groovy code fragment</i>
<pre> reset(); ResourceSet rs = qs.queryResource(docId, "zero-or-one(//package/text())"); if (rs.getSize() == 1) packageName = (String) rs.getResource(0).getContent(); for (ResourceIterator it = qs.queryResource(docId, "//import/text()").getIterator(); it.hasMoreResources();){ addImport((String) it.nextResource().getContent()); } for (ResourceIterator it = qs.queryResource(docId, "/*[localType][not(type)]").getIterator(); it.hasMoreResources();){ XMLResource tr = (XMLResource) it.nextResource(); String localType = (String) qs.query(tr, "exactly-one(localType/text())") .getResource(0).getContent(); String resolvedType = resolve(localType, tr); if (resolvedType == null) throw new TypeResolutionException(localType, "failed to resolve type"); Element node = (Element) tr.getContentAsDOM(); Element tnode = memDoc.createElementNS(JavaRipper.JAVA_NS, "type"); ResourceSet rs2 = qs.query(tr, "localType/@arrayDim"); if (rs2.getSize() == 1) tnode.setAttribute("arrayDim", (String) rs2.getResource(0).getContent()); tnode.appendChild(memDoc.createTextNode(resolvedType)); node.appendChild(tnode); } </pre>	<pre> reset() packageName = doc["zero-or-one(//package/text())"].value doc["//import/text()"].eachValue { addImport(it) } doc["/*[localType][not(type)]"].each { localType = it["exactly-one(localType/text())"].value resolvedType = resolve(localType, targetResource) if (resolvedType == null) throw new TypeResolutionException(localType, "failed to resolve type") it.append { j.type { dims = it["localType/@arrayDim"].value return dims == null ? [] : ["arrayDim" : dims] }.call() [resolvedType] } } </pre>

Figure 5. Comparison of Java and Groovy code

3.3.4. Features

This section contemplates some of the more interesting server components and the challenges they might pose.

Source Code Retrieval In order to incrementally generate UML diagrams, Reef needs timely access to the relevant source code from the repository. Notification of changes can be attained by either event-based (push) or polling (pull) mechanisms, depending on the facilities offered by the repository. All repositories also support change tracking in one form or another, so it should always be possible to obtain at least a coarse-grained list of files changed between dates or versions. To increase adoptability, Reef should be able to interface with a wide selection of repository systems. CruiseControl provides a multitude of simple adapters, though they usually rely on a native installation of the repository's client tools. If installing native clients proves too onerous,

Eclipse (<http://www.eclipse.org/>) has a few adapters that connect directly to the repository server.

The most popular repository system — especially for open-source projects — is CVS, making it the primary target for Reef’s implementation. Much has been written about extracting source code and its history from CVS repositories [ZW04, Ger04, FPG03], though little of it applies to Reef. One major problem researchers have tackled is how to reconstitute atomic Modification Requests from CVS’s non-transactional history log, and match them to bug reports and other documents. Reef, however, only ties its diagrams to a point in time (possibly tagged with a version label); it does not care how the changes are structured. I will probably use some version of the sliding window algorithm to try to ensure that a new diagram is not generated in the middle of a commit, but I can afford to use a large window size since it is not important to accurately separate adjacent commits.

Another issue that is starting to be addressed in literature is how to deal with branching. It is not clear how to link code branches to diagrams and merging can be difficult to detect; I propose to ignore branching in this project unless it becomes unavoidable.

Fact Extraction and Elaboration | Static fact extraction from well-behaved statically typed programming languages is now commonplace, even in commercial tools. Dynamic fact extraction, once the exclusive province of profilers and optimizers, is gaining traction in the reverse engineering arena [GDJ02, HL03a], but few have tried to integrate the two [Tar00]. Since Reef is meant to be used on code under development with unit tests that compile and run — often a chancy proposition for legacy systems — it is in a unique position to advance the state of the art in dynamic fact extraction. To avoid generating unmanageably long traces, I propose at first to use dynamic extraction in a focused fashion, to fill in blanks in the static knowledge base. When the static extractors are unable to ascertain a needed fact (e.g., to infer the type of elements held by a given collection instance [Dug99]), they could request a dynamic trace customized by weaving in aspects specific to their needs [DH+03, Bel04a]. Both the ideas of having co-

operating static and dynamic fact extractors [EKS03b], and of using aspects to instrument code for reverse engineering, are fairly novel. A richer knowledge base should also enable advances in the area of automated design recovery [GA03, KS+02, AFC98, AC+01], saving the developer some effort when raising the diagrams' level of abstraction.

On the fact elaboration front, the identity fuser is a critical component whose function is to track the identity of code-level elements across revisions. Typically, reverse engineering tools perform identity matching solely on the basis of elements' names; this is insufficient for Reef. For example, consider a method that the user has specifically deleted from a diagram. If the method is later renamed, but retains its implementation and relationships to the rest of the code, then it is conceptually the same as its ancestor and should remain deleted in the diagram. Of course, since the identity of an element is merely inferred, it is not possible to be completely certain when it has or has not changed. Nonetheless, many techniques from the burgeoning field of clone detection can be brought to bear, suitably adapted to perform origin analysis [GT02, ZG03] instead. More research is necessary to increase their accuracy, and perhaps integrate a feedback cycle into the algorithms.

Diagram Layout | Good automated diagram layout will be critical to Reef's acceptance: there is little worse than having to "clean up" a large, initially incomprehensible diagram by hand. Though much work has been done on layout algorithms over the years, few researchers have tried to apply general-purpose heuristic global optimization algorithms to the problem, probably because they tend to be computationally demanding. However, thanks to Moore's Law, computing power has grown exponentially over the last few decades, and Reef's architecture places the diagram layout process out of the user's sight, making efficiency less of an issue.

Simulated annealing [KGV83] is one such optimization technique that I have experimented with in the past [Kam02b]. Based on a simple physical process, it basically explores the solution space in a semi-random manner. Though the results are obviously

non-deterministic,¹¹ the algorithm has produced excellent results for the travelling salesman problem and various component and wire layout tasks. The technique was applied to generic graph layout with encouraging results [DH96], and has recently resurfaced as part of a general graph description and layout system [HM+02]. I propose to further investigate the applicability of simulated annealing algorithms to UML diagram layout, perhaps hybridized with genetic algorithms [EM96] and other promising approaches [GJ+03, EKS03a, HL03b].

There is no guarantee that simulated annealing (or other global optimization techniques) can improve the state of the art in UML diagram layout. However, no matter the outcome, an additional result of this effort will be an objective aesthetic metric for UML diagrams that could help in evaluating layout algorithms in the future.

3.4. Front-end Design

Reef's front-end comes in the form of editable diagrams that are embedded in on-line documentation and sent to developers for change ratification. The architectural constraints on the front-end make the choice of platform a primary consideration that, given the state of client-side technologies, may impose considerable limitations on the front-end's other aspects.

3.4.1. Platform

There are few platforms that combine the universal reach of the web browser with the richness of an interface capable of supporting real-time interaction with complex diagrams. Three technologies that play in this space are Macromedia's Flash (<http://www.macromedia.com/software/flash/>), Sun's Java applets (<http://java.sun.com/applets/>) and the Scalable Vector Graphics (SVG) markup language (<http://www.w3.org/Graphics/SVG/>). I evaluate their suitability to the Reef project based on the following criteria:

- *Penetration.* How widespread is the client platform required to run applications?
- *Installation.* How easy is it to set up the client platform if it is missing?

¹¹ Assuming the random number generator is actually random!

- *Storage.* Does the platform allow applications to store data on the client machine?
- *Programming.* Does the platform's programming language provide structural support for large scale programs?
- *Presentation.* Does the platform provide a powerful vector drawing and interaction framework? This can either be part of the platform or a third-party library.
- *Repurposing.* Can users take diagrams and embed them in their own documents with a minimum of work? Can they extract pieces of the diagrams easily? Can the diagrams be indexed by search engines? Can they be printed?
- *Extensibility.* Does the platform allow client-side applications to be customized or extended in a modular fashion? Is it easy for users to add small new pieces of functionality?

While the platforms' performance characteristics are also an issue, there is little data available on the matter and none of it is directly comparable. Nonetheless, the performance of Flash and Java applets should be sufficient, as there are working examples of diagram editors on both platforms. Preliminary experiments indicate that SVG should be able to render diagrams of moderate complexity as well [KWM02]. With some confidence that all three platforms satisfy Reef's basic client-side requirements, let us move on to a more detailed discussion of their pros and cons, as summarized in the adjacent table.

Criterion	Client-side platform		
	<i>Flash</i>	<i>Applets</i>	<i>SVG</i>
<i>Penetration</i>	High	Medium	Low
<i>Installation</i>	Easy	Moderate	Moderate
<i>Storage</i>	High	None	Low+
<i>Programming</i>	Medium	High	Low
<i>Presentation</i>	Medium	Medium+	High+
<i>Repurposing</i>	Low	Low	High
<i>Extensibility</i>	Medium+	Low	High

Flash | Macromedia's Flash is a

mature, nigh-ubiquitous, presentation platform; most browsers have the plug-in installed, but if not the download clocks in at a svelte 480Kb. However—especially among open-source developers, part of the target audience for Reef—Flash has a reputation as a toy for displaying annoying ads, with some people pointedly refusing to in-

stall it. This reputation is not completely undeserved, but Flash is certainly more than a toy. In its latest MX 2004 version, Flash sports ActionScript 2.0—an implementation of the perpetually in progress ECMAScript Edition 4 standard [Hor03], which adds class-oriented programming features to the familiar untyped prototype-based 3rd edition [EC+99]. In support of browser-hosted applications, the standard libraries provide mechanisms for communicating with the server in XML, and Flash gives applications access to as much local storage as they desire (subject to the user’s approval).

Flash also features a reasonably complete vector rendering and interaction engine. Although only line and quadratic spline primitives are exposed to ActionScript, it is possible to build up sophisticated user interfaces, such as the impressive gModeler (<http://www.gskinner.com/gmodeler/>)—an all-Flash UML class diagram editor. Regrettably, applications are packaged into opaque binary blobs that make it difficult to dynamically bundle the data or extend the code, but there are possible bridges from XML (e.g., KineticFusion (<http://www.kinesisssoftware.com/>) or the very expensive Macromedia Flex (<http://www.macromedia.com/software/flex/>)). A Flash application’s canvas is also not easily exportable, limiting the ability of users to repurpose or extend the diagrams.

Applets | Java is an industrial-strength object-oriented programming language, making it an excellent choice for Reef’s back-end. On the client side, however, Sun’s vision of Java applets never really caught on. The Java virtual machine is available in many, but by no means all browser installations, and it is rarely the latest version of the JDK. We can hope that with the recent rapprochement between Sun and Microsoft, more recent versions of the virtual machine will be bundled with Windows, but in the meantime the required download weighs in at a whopping 14.6Mb.

There are many vector drawing libraries available for Java, including the excellent Piccolo (<http://www.cs.umd.edu/hcil/piccolo/>), which I happen to be familiar with [CC+03]. However, Java applets suffer from the same problems as Flash applications: the live vector drawing is not easily exportable, as it is highly dependent on the applet code, and writing extensions to an applet is a high-ceremony affair, unless the applet integrates some

kind of dynamic code interpreter. Worse, Java applets have no access to local storage unless they are signed, but cross-browser applet signing is a tricky proposition. Java applets are not a good fit for Reef's client-side requirements.

SVG | SVG is a new vector graphics language designed and promulgated by the W3C. SVG 1.1 [FFJ03] has achieved some small measure of success with developers, but suffers from a low install base even though a browser plug-in is bundled by default with downloads of Acrobat Reader (version 5 and higher). (The Adobe SVG viewer can also be downloaded separately, weighing in at a reasonable 2.3Mb.) The W3C is also working on introducing many significant improvements to SVG 1.2 [Jac04]—some of which are mentioned below—but upon release the install base will have to restart from scratch.

SVG 1.1 offers a dizzying array of vector graphics and declarative animation primitives, all expressed in easy to repurpose XML. SVG 1.2 improves support for flowing and editing text, and introduces the Rendering Custom Content (RCC) facility¹² [Qui03] for declaratively specifying an SVG binding to an arbitrary XML vocabulary. SVG 1.2 also adds access to a small amount of local storage, but this capability can be emulated in SVG 1.1 through clever (ab)use of browser cookies, taking advantage of the integration between the SVG plug-in and Internet Explorer. Of course, this forces the client-side data to be small; see Section 3.4.2 for details.

SVG (both 1.1 and 1.2) is powered by 3rd edition ECMAScript [EC+99], making it eminently extensible. This prototype-based object-oriented scripting language is reasonably powerful, with first-class functions and closures, but provides no built-in mechanisms for building large programs, such as encapsulation and namespace management. To prevent the diagram editor code from quickly becoming unmaintainable [Gre04], it should be possible to leverage the basic ECMAScript facilities into more expressive constructs. The *jsolait* library (<http://jan.kollhof.net/projects/js/jsolait/>) makes initial efforts along these lines, but many other improvements beckon. The security concerns addressed by

¹² It looks like RCC will be extracted from SVG and merge with the XML Binding Language (XBL) [Hya01] into its own project in the near future. The functionality is similar to that provided by Microsoft's HTML behaviours [Wil98].

capabilities [MS03] as implemented in the E language (<http://www.erights.org/>) may inspire an encapsulation strategy, and aspect-oriented and traits-based [SD+03, OA+04] programming may help impose some structure on the language's primordial prototype system.

Verdict | Using Java applets for Reef is infeasible (mainly due to the lack of client-side storage), but both Flash and SVG (especially the upcoming version 1.2) look appealing. They have complimentary characteristics: Flash is mature and widespread but old and proprietary, while SVG is XML-based and open but virtually untested. I will initially go with SVG 1.2, as it appears to be the way of the future and, as a new technology, may provide fertile ground for interesting implementation-level research. However, should being a pioneer prove too time-consuming or if SVG fails to deliver on its promises, I will switch over to Flash.

3.4.2. Data Management and Communication

The nature of a zero-install client component embedded in an email message poses some challenges to data management and communication. Once a diagram is received by the user, where should updates be stored? How should they be sent back to the server, and possible concurrent changes reconciled? Answers to these questions will have a direct impact on adoptability.

Since the client is likely to have restricted local storage access privileges, and since it is inconvenient to reconcile diagrams based solely on the full documents, I intend to store updates as a compacted edit list. Not only does this approach save space, but it also allows users to send lightweight diagram “patches” to each other and enables the edits to be reconciled using algorithms derived from real-time collaborative editing tools [Cor95, SE98]. The same delta format can be used to commit updates to the server, with authorization based on the authentication provided by the transmission protocol (e.g., cryptographic email signatures). A uniform delta storage and transmission format allows for code reuse, reducing the client component's size. However, the delta storage

technique relies on edit lists being short, as they must be re-applied to the base model every time an uncommitted diagram is opened by the user.

Due to the nature of the client, it is also impossible to include in the component all data that may be of interest (e.g., past versions of a diagram) and to perform some computationally intensive operations locally (e.g., a full diagram layout). In situations where the server's assistance is required, the client should first attempt to establish a direct connection to the server, but fall back gracefully on asynchronous communication protocols (e.g., email) if necessary.

3.4.3. User Interface

Figure 6 displays an initial mock-up of Reef's user interface, as a visual reference for the detailed feature explanations that follow. The diagram editor is shown running inside a browser, though it could just as well run inside a mail client (subject to circumventing excessive security restrictions).

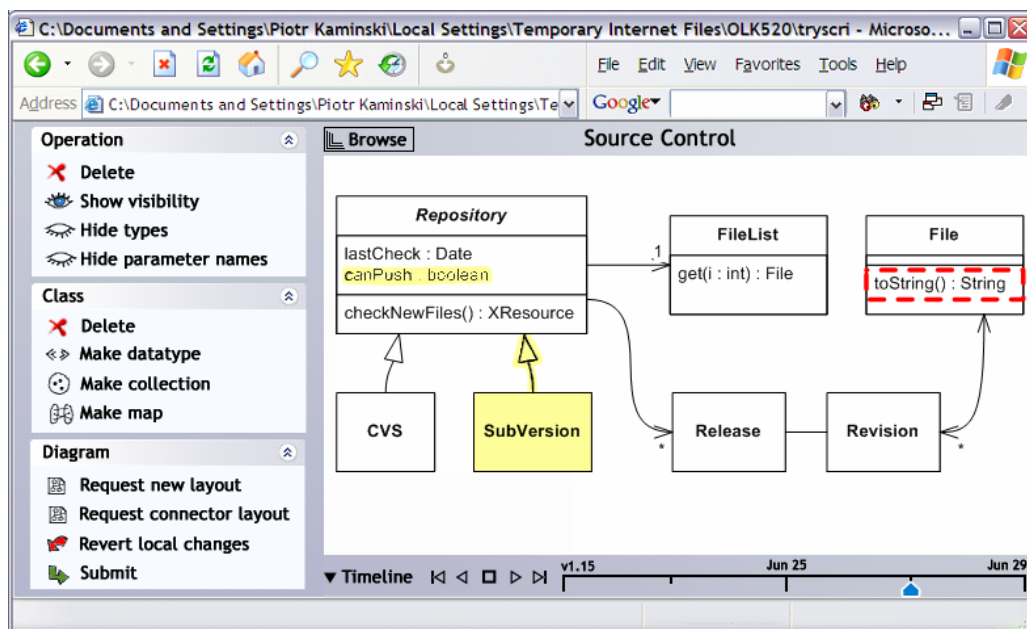


Figure 6. Client user interface mock-up

General Principles | The main interface area displays the UML diagram, with a title providing context. The UML diagrams largely follow established OMG standards [OMG03], but do not hesitate to depart from them if a popular notation variant is more

readable. Conversely, not all notational variants are offered, even if sanctioned by the standard; rather, only a coherent subset is made available to the user. Taking such freedoms with the standard is consistent with the “UML as sketch” perspective [FowA] and makes the tool more prescriptive in flavour, to try to encourage developers to construct good diagrams [MM03]. In this vein, I am considering disabling both the zooming and scrolling facilities normally found in editors, to force the diagrams into a reasonable size suitable for effective communication. Use of fisheye distortion [Bed00, JM03] could compensate for the inadequate resolution of current screen technologies.

There are two global interface modes that affect how the diagram responds to user actions. In browse mode, clicking on an element offers links to related information (e.g., other diagrams that contain this element, Javadocs, source code, etc.). In edit mode, dragging an element moves and resizes it, while clicking one selects it as the target for context-sensitive commands displayed on the left. In Figure 6, the `toString()` operation of the `File` class is selected, and relevant commands for the operation element and its ancestors are listed in the left column. This “taskbar” approach, similar to the one used in Windows XP Explorer, flattens the learning curve and enables casual use of the tool,¹³ while experts can customize the hotkey bindings. Naturally, all commands applied to the diagram can be undone (to some reasonable depth), and a diagram can always be reverted to its original form thanks to the delta storage mechanism.

Due to Reef’s workflow design (see (U1)), we can safely assume that the vast majority of a diagram’s contents will have been generated via reverse-engineering, leaving the user to refine and fine-tune the diagram. For this reason, Reef’s client-side command set concentrates on alteration rather than creation¹⁴ (see Appendix A for a sample list of commands for class diagrams). For example, combining a pair of read/write accessors into an attribute is a single action in Reef, whereas in typical diagram editors the user

¹³ My observations of casual Windows XP users indicate that they prefer to use the taskbar even if the same commands are available in a right-click context menu. This is true even of users that have learned much more complex and efficient user interfaces in their specialized applications.

¹⁴ This might go as far as not letting the user create new diagram elements at all, though such a restriction might not prove workable in practice.

would delete the two operations and create a new attribute manually. Not only will such commands speed the editing process, but they will also maintain traceability to the underlying implementation-level elements, allowing further automation of diagram maintenance. For example, should the data type of the accessors mentioned above change, Reef could automatically update the type of the corresponding “virtual” attribute; this would not be possible with the manual delete/create approach.

Action Amplification Even with a rich set of refinement-oriented commands, touching up every new element of an updated diagram can be very repetitive. To alleviate the tedium, Reef offers an action amplification mechanism. After applying a command to an element, the user is given the chance to amplify its effects over the element’s container, the diagram, all diagrams in the current project, or all diagrams in the repository, as appropriate. Not only is the action’s target generalized and the command immediately re-applied, but the amplified action is kept by Reef and automatically applied to new elements that match the pattern as they get created.

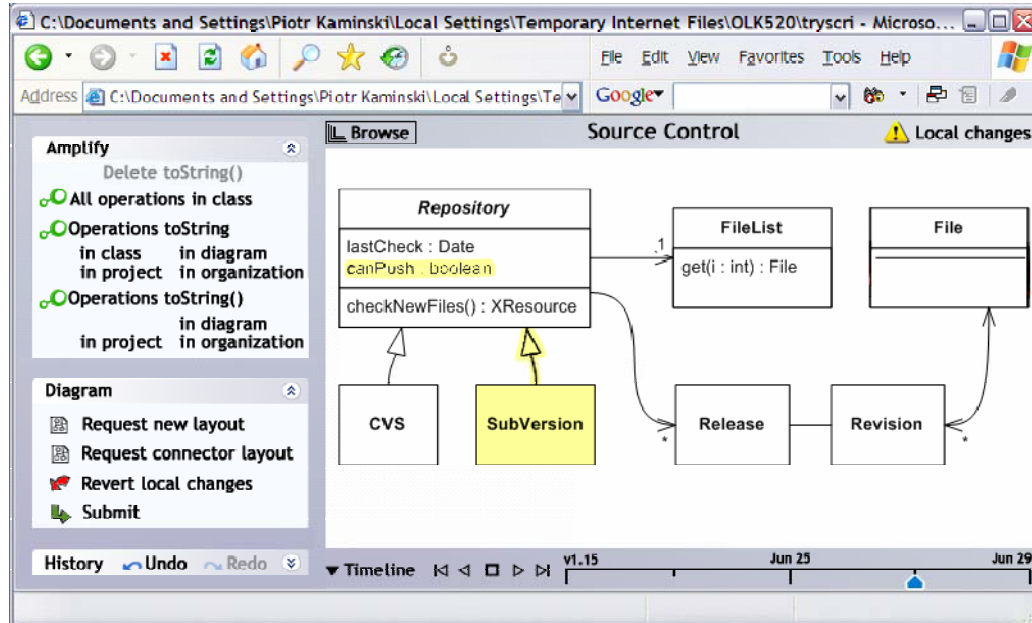


Figure 7. Action effect and amplification mock-up

For example, consider Figure 7: the `toString()` operation in the `File` class was just deleted. The Amplify box in the left column now proposes various ways to expand the scope of the

deletion. The user can choose to delete all operations in the File class; though this would have no immediate effect (as there are no operations remaining in this class), any future operations added by reverse-engineering code would be automatically deleted by Reef before ever being shown to the user. Similarly, if the user chose to amplify the action to all `toString()` operations in the project, all current and future `toString()` operations would be deleted automatically.

Potential amplifiers are specific to each command; Appendix A gives some examples for class diagram commands. Simple amplifiers, such as applying “hide operation signature” to the whole diagram, take the place of separate preference options common to other tools. For example, it is often possible to hide some feature of an element through a pop-up menu, but to hide the same feature in the whole diagram requires finding a separate preferences dialog box; in this case, action amplification becomes a sort of “preferences by example”. On the other hand, more complex effects are possible. If Reef notices that the user always deletes operations that start with the word “test”, next time it might offer to amplify the action to delete all operations that match the pattern “test*”. The user could also enter arbitrary selection patterns using the full XPath language, making for a powerful facility with a gentle learning curve.

Although action amplification could greatly speed up touch-up of incrementally generated diagrams, there are potential pitfalls. The wider an action’s scope, the better the chance of finding a situation in which it is not actually applicable. There must be a way for the user to examine the amplified actions affecting an element (even a deleted one!) and make exceptions. Ultimately, action amplification is a general declarative diagram transformation facility, and as with all such services, a balance must be reached between the time saved through automation and the time spent on maintaining the rule base.

Evolution Animation | When a developer receives an updated UML diagram for review, it is critical to attract his attention to the automated changes caused by Reef’s incremental reverse-engineering process. Differences between two versions of a UML diagram can be highlighted with appropriate use of color and line styles (see Figure 8), but the technique does not scale to longer series of diagrams. Since consistent use of Reef may produce as much as one new diagram version for each source repository commit—perhaps one per day!—the tool could benefit from a different approach.

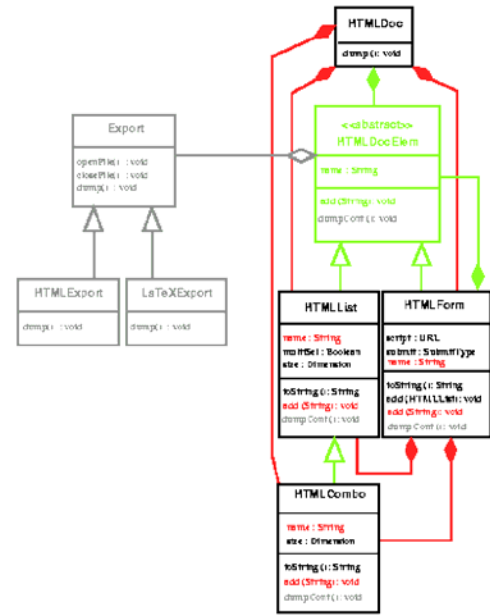


Figure 8. Stylistic UML diff [OWK03]

My idea is to put a user-selectable subset of a diagram’s versions on a timeline, labelled by their timestamps or source code version numbers (if available). Half way between each consecutive pair of diagrams, there is an intermediate pair wise difference diagram that uses stylistic conventions to indicate modifications (e.g., yellow highlights for new elements in Figure 6). The transitions between all the diagrams on the timeline are animated; elements move, change styles and fade in and out to compose an intermediate diagram, and then again to reach a stamped checkpoint. The user can control which diagrams are shown on the timeline (requesting additional versions from the server as necessary), and can play the animation back and forth, or scrub through it manually.

Animation is often used to help users follow state changes in an application. For Reef, I hope that animating design diagram transitions will allow users to intuitively perceive patterns of change in the system that are impossible to identify algorithmically and that would get lost in the noise of a purely stylistic comparison. Nonetheless, this is an experimental feature that will need to be tested and refined in real-world conditions, with no guarantee of success.

4. Conclusions

This section presents my initial research plan and the results I expect to obtain over the course of the project.

4.1. *Research Plan*

The research plan is straightforward (time estimates in parenthesis):

1. Feasibility study and initial literature survey (completed).
2. End-to-end proof of concept (3 months).
3. Tool development: static extractor and diagrams (5 months).
4. Tool development: dynamic extractor and diagrams (5 months).
5. Empirical and analytical evaluation (8 months).

The end-to-end proof of concept will implement the high-risk base framework of the tool to demonstrate a full cycle as described in (U1) and (U2). When the static portion of the tool is completed (at the end of step 3), I intend to release it to a wider audience to gather initial feedback while I work on the dynamic parts. Evaluation of the tool and testing of selected hypotheses is scheduled for the end of the project, and takes into account the time necessary to set up empirical studies.

4.2. *Expected Contributions*

The Reef project is rife with opportunities to significantly advance the state of the art [MJ+00] in a number of areas. I expect to:

1. Provide evidence towards some of the hypotheses (H1) through (H6) and (H9) via both controlled and natural human experiments, the latter to avoid the Hawthorne effect [May46] on performance studies and attempt to increase their external validity.
2. Introduce new approaches to diagram editor user interfaces: a focus on refinement rather than wholesale creation, evolution animation, and action amplification.

3. Innovate in the fields of full and incremental off-line diagram layout using heuristic global optimization techniques.
4. Discover patterns and document best practices for building complex SVG and ECMAScript (3rd edition) applications, and potentially codify my findings in a framework.
5. Improve reverse engineering performance by better integrating static and dynamic methods and enhancing origin analysis algorithms.
6. Experiment with XML databases and evaluate their usefulness as fact stores for software modeling projects.

I also hope that the Reef tool itself will transcend its genesis as a research vehicle and become an important component of developers' toolkits world-wide.

Bibliography

All references to W3C documents list the URI of the “latest version”, since the URI is shorter and the most recent revision of a recommendation is likely to be of more interest than the exact one in effect at the time of writing of this thesis. However, should you wish to see this older version, you can follow the link from the current document to the older one identified by the publication date given in the reference.

All references to OMG documents list the version 1.x specifications. Versions 2.0 of the MOF, UML and XMI are expected to be completed soon, but were not yet released at the time of this writing. Nonetheless, I intend to use the version 2.0 specifications for this project.

- [AC+01] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, Narendra Jussien. *Instantiating and Detecting Design Patterns*. In Proceedings of the 16th International Conference on Automated Software Engineering, p. 166, November 2001.
- [AFC98] G. Antoniol, R. Fiutem, L. Cristoforetti. *Design Pattern Recovery in Object-Oriented Software*. In Proceedings of the 6th International Workshop on Program Comprehension, p. 153, June 1998.
- [Amb03a] Scott W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2003. ISBN 0521525470
- [Amb03b] Scott W. Ambler. *Agile Model Driven Development Is Good Enough*. IEEE Software, volume 20, issue 5, pp. 71-73, September/October 2003. [doi:10.1109/MS.2003.1231156](https://doi.org/10.1109/MS.2003.1231156)
- [AS03] Ritu Agarwal, Atish P. Sinha. *Object-oriented Modeling with UML: A Study of Developers' Perceptions*. Communications of the ACM, volume 46, issue 9, pp. 248-256, September 2003. [doi:10.1145/903893.903944](https://doi.org/10.1145/903893.903944)
- [ASB99] Serge Abiteboul, Dan Suciu, Peter Buneman. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, October 1999. ISBN 155860622X
- [BB+03] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, Jérôme Siméon, eds. *XML Path Language (XPath) 2.0*. W3C Working Draft, November 2003. <http://www.w3.org/TR/xpath20/>
- [BC+03] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, eds. *XQuery 1.0: An XML Query Language*. W3C Working Draft, November 2003. <http://www.w3.org/TR/xquery/>
- [Bed00] Benjamin B. Bederson. *Fisheye Menus*. In Proceedings of the 13th Symposium on User Interface Software and Technology, pp. 217-225, November 2000. [doi:10.1145/354401.354782](https://doi.org/10.1145/354401.354782)

- [Bel04a] Abhijit Belapurkar. *Use AOP to maintain legacy Java applications*. IBM developerWorks, March 2004. <http://www-106.ibm.com/developerworks/library/j-aopsc2.html?ca=drs-t511>
- [Bel04b] Alex E. Bell. *Death by UML Fever*. ACM Queue, volume 2, number 1, March 2004. doi:10.1145/984458.984495
- [BHL99] Tim Bray, Dave Hollander, Andrew Layman, eds. *Namespaces in XML*. W3C Recommendation, January 1999. <http://www.w3.org/TR/REC-xml-names/>
- [BJ+03] Robert Balzer, Jens-Holger Jahnke, Marin Litoiu, Hausi A. Müller, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, Kenny Wong, Anke Weber. 3rd International Workshop on Adoption-Centric Software Engineering, June 2003.
- [Bla00] Steven Black. *Documentation Timing*. FoxPro Wiki, August 2000. <http://fox.wikis.com/wc.dll?Wiki~DocumentationTiming~SoftwareEng>
- [Bou04] Ronal Bourret. *XML Database Products*. May 2004. <http://www.rpbourret.com/xml/XMLDatabaseProds.htm>
- [BPS00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, eds. *Extensible Markup Language (XML) 1.0, 3rd ed*. W3C Recommendation, February 2004. <http://www.w3.org/TR/REC-xml>
- [CC+03] James Chisan, Jeff Cockburn, Reid Garner, Azarin Jazayeri, Piotr Kaminski, Jesse Wesson. *Video Bench, Final Report*. Project report for CSc 586a, University of Victoria, April 2003.
- [Cor89] Thomas A. Corbi. *Program Understanding: Challenge for the 1990s*. IBM Systems Journal, volume 28, number 2, pp. 294-306, 1989.
- [Cor95] Gordon V. Cormack. *A Calculus for Concurrent Update*. University of Waterloo, Research Report CS-95-06, 1995.
- [CTM03] Stuart M. Charters, Nigel Thomas and Malcolm Munro. *The end of the line for Software Visualisation?* In Proceedings of the 2nd Workshop on Visualizing Software for Analysis and Understanding, September 2003.
- [DDL99] Serge Demeyer, Stéphane Ducasse, Michele Lanza. *A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualisation*. In Proceedings of the 6th Working Conference on Reverse Engineering, pp. 175-186, October 1999.
- [DH+03] Jonathan Davies, Nick Huismans, Rory Slaney, Sian Whiting, Matthew Webster. *An Aspect Oriented Performance Analysis Environment*. Practitioners' Report from the International Conference on Aspect-Oriented Software Development, March 2003.
- [DH96] Ron Davidson, David Harel. *Drawing Graphs Nicely Using Simulated Annealing*. ACM Transactions on Graphics, volume 15, issue 4, pp. 301-331, October 1996. doi:10.1145/234535.234538
- [Dug99] Dominic Duggan. *Modular type-based reverse engineering of parameterized types in Java code*. In Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages and Applications, pp. 97-113, November 1999. doi:10.1145/320384.320393

- [DV02] Péter Domokos, Dániel Varró. *An Open Visualization Framework for Metamodel-Based Modeling Languages*. In Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2002), pp. 78-87, October 2002.
- [EC+99] ECMA. *ECMAScript Language Specification, 3rd edition*. Standard ECMA-262, December 1999.
- [EG03] Holger Eichelberger, Jurgen Wolff von Gudenberg. *UML Class Diagrams – State of the Art in Layout Techniques*. In Proceedings of the 2nd Workshop on Visualizing Software for Understanding and Analysis, September 2003.
- [Eic02a] Holger Eichelberger. *Aesthetics of Class Diagrams*. In Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis, pp. 23-31, June 2002. doi:[10.1109/VISSOF.2002.1019791](https://doi.org/10.1109/VISSOF.2002.1019791)
- [Eic02b] Holger Eichelberger. *Evaluation-Report on the Layout Facilities of UML Tools*. Technical Report number 298, University of Würzburg, July 2002.
- [Eic03] Holger Eichelberger. *Nice Class Diagrams Admit Good Design?* In Proceedings of the 2003 ACM Symposium on Software Visualization, p. 159, June 2003. doi:[10.1145/774833.774857](https://doi.org/10.1145/774833.774857)
- [EKS03a] Markus Eiglsperger, Michael Kaufmann, Martin Siebenhaller. *A topology-shape-metrics approach for the automatic layout of UML class diagrams*. In Proceedings of the Symposium on Software Visualization, p. 189, June 2003. doi:[10.1145/774833.774860](https://doi.org/10.1145/774833.774860)
- [EKS03b] Thomas Eisenbarth, Rainer Koschke, Daniel Simon. *Locating Features in Source Code*. IEEE Transactions on Software Engineering, volume 29, number 3, pp. 210-224, March 2003. doi:[10.1109/TSE.2003.1183929](https://doi.org/10.1109/TSE.2003.1183929)
- [EM96] Timo Eloranta, Erkki Mäkinen. *TimGA: A Genetic Algorithm for Drawing Undirected Graphs*. Technical Report A-1996-10, University of Tampere, 1996.
- [FFJ03] Jon Ferraiolo, Jun Fujisawa, Dean Jackson, eds. *Scalable Vector Graphics (SVG) 1.1 Specification*. W3C Recommendation, January 2003. <http://www.w3.org/TR/SVG11/>
- [Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000. ISBN 0201485672
- [Fow04] Martin Fowler. *UML Distilled, Third Edition*. Addison-Wesley, 2004. ISBN 0321193687
- [FowA] Martin Fowler. *UmlAsSketch*. Bliki entry, undated. <http://www.martinfowler.com/bliki/UmlAsSketch.html>
- [FPG03] Michael Fischer, Martin Pinzger, Harald Gall. *Populating a Release History Database from Version Control and Bug Tracking Systems*. In Proceedings of the International Conference on Software Maintenance, pp. 23-32, September 2003.
- [GA03] Yann-Gaël Guéhéneuc, Hervé Albin-Amiot. *A Pragmatic Study of Binary Class Relationships*. In Proceedings of the 18th Conference on Automated Software Engineering, September 2003. doi:[10.1109/ASE.2003.1240320](https://doi.org/10.1109/ASE.2003.1240320)

- [GDJ02] Yann-Gaël Guéhéneuc, Rémi Douence, Narendra Jussien. *No Java without Caffeine*. In Proceedings of the 17th IEEE International Conference on Automated Software Engineering, pp. 117-126, 2002. [doi:10.1109/ASE.2002.1115000](https://doi.org/10.1109/ASE.2002.1115000)
- [Ger04] Daniel M. German. *Mining CVS Repositories, the softChange Experience*. In Proceedings of the International Workshop on Mining Software Repositories, May 2004.
- [GJ+03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, Petra Mutzel. *A New Approach for Visualizing UML Class Diagrams*. In Proceedings of the 2003 ACM Symposium on Software Visualization, pp. 179-188, June 2003. [doi:10.1145/774833.774859](https://doi.org/10.1145/774833.774859)
- [Gre04] Roedy Green. *How To Write Unmaintainable Code*. Last updated June 2004. <http://mindprod.com/unmain.html>
- [GT02] Michael Godfrey, Qiang Tu. *Tracking Structural Evolution using Origin Analysis*. In Proceedings of the 2002 International Workshop on Principles of Software Evolution, May 2002. [doi:10.1145/512035.512062](https://doi.org/10.1145/512035.512062)
- [HL03a] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge. *Techniques for Reducing the Complexity of Object-Oriented Execution Traces*. In Proceedings of the 2nd Annual Designfest on Visualizing Software for Understanding and Analysis, pp. 35-40, October 2003.
- [HL03b] Xiaodi Huang, Wei Lai. *Force-Transfer: A New Approach to Removing Overlapping Nodes in Graph Layout*. In Proceedings of the 26th Australasian Computer Science Conference on Research and Practice in Information Technology, volume 16, pp. 349-358, 2003.
- [HLN04] Jouni Huotari, Kalle Lyytinen, Marketta Niemelä. *Improving graphical information system model use with elision and connecting lines*. ACM Transactions on Computer-Human Interaction, Volume 11, Issue 1, pp. 26-58, March 2004. [doi:10.1145/972648.972650](https://doi.org/10.1145/972648.972650)
- [HM+02] Trevor Hansen, Kim Marriott, Bernd Meyer, Peter J. Stuckey. *Flexible Graph Layout for the Web*. Journal of Visual Languages and Computing, volume 13, issue 1, pp. 35-60, 2002. [doi:10.1006/jvlc.2001.0226](https://doi.org/10.1006/jvlc.2001.0226)
- [Hol02] Allen Holub. *When it comes to good OO design, keep it simple*. JavaWorld, January 2002. <http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-ootools.html>
- [Hor03] Waldemar Horwat. *ECMAScript 4 Netscape Proposal*. June 2003. <http://www.mozilla.org/js/language/es4/>
- [Hya01] David Hyatt. *XBL – XML Binding Language*. W3C Note, February 2001. <http://www.w3.org/TR/xbl/>
- [Jac04] Dean Jackson, ed. *Scalable Vector Graphics (SVG) 1.2*. W3C Working Draft, May 2004. <http://www.w3.org/TR/SVG12/>
- [JM03] Timothy Jacobs, Benjamin Musial. *Interactive Visual Debugging with UML*. In Proceedings of the 2003 ACM Symposium on Software Visualization, pp. 115-122, June 2003. [doi:10.1145/774833.774850](https://doi.org/10.1145/774833.774850)

- [Joh03] Philip M. Johnson. *Results from Qualitative Evaluation of Hackystat-UH*. Technical Report number CSDL-03-13, Department of Information and Computer Sciences, University of Hawaii, December 2003.
- [Kam02a] Piotr Kaminski. *Integrating Information on the Semantic Web Using Partially Ordered Multi Hypersets*. MSc Thesis, University of Victoria, September 2002.
- [Kam02b] Piotr Kaminski. *Las Vegas Go*. CSc 581a project report, December 2002.
- [KC04] Graham Klyne, Jeremy J. Carroll, eds. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, February 2004. <http://www.w3.org/TR/rdf-concepts/>
- [Kem99] Eva Kemps. *Effects of Complexity on Visuo-spatial Working Memory*. European Journal of Cognitive Psychology, volume 11, number 3, pp. 335-356, September 1999.
- [KG01] Ralf Kollmann, Martin Gogolla. *Capturing Dynamic Program Behaviour with UML Collaboration Diagrams*. In Proceedings of the 5th European Conference on Software Maintenance and Reengineering, pp. 58-67, March 2001. [doi:10.1109/CSMR.2001.914969](https://doi.org/10.1109/CSMR.2001.914969)
- [KG02] Ralf Kollmann, Martin Gogolla. *Metric-Based Selective Representations of UML Diagrams*. In Proceedings of the 6th European Conference on Software Maintenance and Reengineering, pp. 89-98, March 2002. [doi:10.1109/CSMR.2002.995793](https://doi.org/10.1109/CSMR.2002.995793)
- [KG04] Cory Kapser, Michael W. Godfrey. *Aiding Comprehension of Cloning Through Categorization*. Submitted to International Workshop on Software Evolution, September 2004.
- [KGV83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. *Optimization by Simulated Annealing*. Science, volume 220, number 4598, May 1983.
- [KLW95] Michael Kifer, Georg Lausen, James Wu. *Logical Foundations of Object-Oriented and Frame-Based Languages*. Journal of the ACM, volume 42, pp. 741-783, 1995. [doi:10.1145/210332.210335](https://doi.org/10.1145/210332.210335)
- [KS+02] Ralf Kollmann, Petri Selonen, Eleni Stroulia, Tarja Systä, Albert Zündorf. *A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering*. In Proceedings of the 9th Working Conference on Reverse Engineering, p. 22, November 2002.
- [KWM02] Holger M. Kienle, Anke Weber, Hausi A. Müller. *Leveraging SVG in the Rigi Reverse Engineering Tool*. In Proceedings of the SVG Open, July 2002.
- [May46] Elton Mayo. *The Human Problems of an Industrial Civilization*.
- [MC+02] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, Rosemary Monahan. *Reveal: A Tool to Reverse Engineer Class Diagrams*. In Proceedings of the 40th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications, volume 10, pp. 13-21, February 2002.
- [MJ+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, Kenny Wong. *Reverse Engineering: A Roadmap*. In The Future of Software Engineering (Anthony Finkelstein, ed.), ACM Press, 2000. ISBN 1581132530

- [MM03] Neil MacKinnon, Steve Murphy. *Designing UML Diagrams for Technical Documentation*. In Proceedings of the 21st Annual International Conference on Documentation, pp. 105-112, October 2003. [doi:10.1145/944868.944891](https://doi.org/10.1145/944868.944891)
- [MS03] Mark S. Miller, Jonathan S. Shapiro. *Paradigm Regained: Abstraction Mechanisms for Access Control*. In Proceedings of the 8th Asian Computing Science Conference, December 2003.
- [MV95] Anneliese von Mayrhauser, A. Marie Vans. *Program Comprehension During Software Maintenance and Evolution*. IEEE Computer, volume 28, issue 8, pp. 44-55, August 1995. [doi:10.1109/2.402076](https://doi.org/10.1109/2.402076)
- [NL03] Colin J. Neill, Phillip A. Laplante. *Requirements Engineering: The State of the Practice*. IEEE Software, volume 20, number 6, pp. 40-45, November/December 2003. [doi:10.1109/MS.2003.1241365](https://doi.org/10.1109/MS.2003.1241365)
- [OA+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger. *An Overview of the Scala Programming Language*. Draft, June 2004.
- [OMG02a] Object Management Group. *Meta Object Facility (MOF) Specification*. Version 1.4, April 2002.
- [OMG02b] Object Management Group. *XML Metadata Interchange (XMI) Specification*. Version 1.2, January 2002.
- [OMG03] Object Management Group. *OMG Unified Modeling Language Specification*. Version 1.5, March 2003.
- [OWK03] Dirk Ohst, Michael Welle, Udo Kelter. *Differences Between Versions of UML Diagrams*. In Proceedings of the 9th European Software Engineering Conference, pp. 227-236, September 2003. [doi:10.1145/940071.940102](https://doi.org/10.1145/940071.940102)
- [PAC00] Helen C. Purchase, Jo-Anne Alder, David A. Carrington. *User Preference of Graph Layout Aesthetics: A UML Study*. In Proceedings of the 8th International Symposium on Graph Drawing, pp. 5-18, September 2000.
- [Pas04] Fabian Pascal. *If You Liked SQL, You'll Love XQuery*. DBAzone.com, June 2004. <http://www.dbazine.com/pascal19.shtml>
- [PC+01] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington, Carol Britton. *UML Class Diagram Syntax: An Empirical Study of Comprehension*. In Proceedings of the Australian Symposium on Information Visualisation, volume 9, pp. 113-120, 2001.
- [PC+02] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington. *UML Collaboration Diagram Syntax: An Empirical Study of Comprehension*. In Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis, pp. 13-22, June 2002. [doi:10.1109/VISSOF.2002.1019790](https://doi.org/10.1109/VISSOF.2002.1019790)
- [PM+01] Helen C. Purchase, Matthew McGill, Linda Colpoys, David Carrington. *Graph Drawing Aesthetics and the Comprehension of UML Class Diagrams: An Empirical Study*. In Proceedings of the Australian Symposium on Information Visualisation, volume 9, pp. 129-137, 2001.

- [Pur00] Helen C. Purchase. *Effective information visualisation: a study of graph drawing aesthetics and algorithms*. Interacting with Computers, volume 12, issue 2, pp. 147-162, December 2000. [doi:10.1016/S0953-5438\(00\)00032-1](https://doi.org/10.1016/S0953-5438(00)00032-1)
- [Qui03] Antoine Quint. *SVG and XForms: Rendering Custom Content*. IBM developerWorks, November 2003. <http://www-106.ibm.com/developerworks/xml/library/x-svgxf2/>
- [RRK99] Daniel H. Robinson, Sheri L. Robinson, Andrew D. Katayama. *When Words Are Represented in Memory Like Pictures: Evidence for Spatial Encoding of Study Materials*. Contemporary Educational Psychology, volume 24, issue 1, pp. 38-54, January 1999. [doi:10.1006/ceps.1998.0979](https://doi.org/10.1006/ceps.1998.0979)
- [Rug94] Spencer Rugaber. *White Paper on Reverse Engineering*. Unpublished, 1994.
- [SD+03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, Andrew P. Black. *Traits: Composable Units of Behaviour*. In Proceedings of the European Conference on Object-Oriented Programming, July 2003.
- [SE98] Chengzheng Sun, Clarence Ellis. *Operational transformation in real-time group editors: issues, algorithms, and achievements*. In Proceedings of the Conference on Computer Supported Cooperative Work, pp. 59-68, November 1998. [doi:10.1145/289444.289469](https://doi.org/10.1145/289444.289469)
- [Sha03] Mary Shaw. *Writing Good Software Engineering Research Papers*. In Proceedings of the 25th International Conference on Software Engineering, pp. 726-736, May 2003.
- [Shi98] Clay Shirky. *View Source... Lessons from the Web's massively parallel development*. April 1998. http://www.shirky.com/writings/view_source.html
- [Sho04] Keith Short. *UML and DSLs Again*. April 2004. http://blogs.msdn.com/keith_short/archive/2004/04/16/114960.aspx
- [Spi03] Diomidis Spinellis. *On the Declarative Specification of Models*. IEEE Software, volume 20, number 2, pp. 94-96, March/April 2003. [doi:10.1109/MS.2003.1184181](https://doi.org/10.1109/MS.2003.1184181)
- [Sta84] T. Standish. *An essay on software reuse*. IEEE Transactions on Software Engineering, volume 10, number 5, pp. 494-497, September 1984.
- [Tar00] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Systems*. Academic Dissertation, University of Tampere, Finland, May 2000.
- [TH03] Scott Tilley, Shihong Huang. *A Qualitative Assessment of the Efficacy of UML Diagrams as a Form of Graphical Documentation in Aiding Program Understanding*. In Proceedings of the 21st Annual International Conference on Documentation, pp. 184-191, October 2003. [doi:10.1145/944868.944908](https://doi.org/10.1145/944868.944908)
- [Try97] Eirik Tryggeseth. *Report from an Experiment: Impact of Documentation on Maintenance*. Journal of Empirical Software Engineering, volume 2, number 2, pp. 201-207, 1997. [doi:10.1023/A:1009778023863](https://doi.org/10.1023/A:1009778023863)
- [Tuf97] Edward R. Tufte. *Visual explanations : images and quantities, evidence and narrative*. Graphics Press, 1997. ISBN 0961392126
- [Vis97] Giuseppe Visaggio. *Relationships between Documentation and Maintenance Activities*. In Proceedings of the 5th International Workshop on Program Comprehension, p. 4, May 1997.

- [Wil98] Chris Wilson, ed. *HTML Components: Componentizing Web Applications*. W3C Note, October 1998. <http://www.w3.org/TR/NOTE-HTMLComponents>
- [Win01] Andreas Winter. *Exchanging Graphs with GXL*. In Proceedings of 9th International Symposium on Graph Drawing, Vienna, September 2001.
- [Won98] Kenny Wong. *Rigi User's Manual, Version 5.4.4*. University of Victoria, June 1998. <http://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf>
- [Zei02] Alan Zeichick. *Modeling Usage Low; Developers Confused About UML 2.0, MDA*. SD Times, July 15, 2002. <http://www.sdtimes.com/news/058/story3.htm>
- [ZG03] Lijie Zou, Michael W. Godfrey. *Detecting Merging and Splitting using Origin Analysis*. In Proceedings of the Working Conference on Reverse Engineering, November 2003.
- [ZW04] Thomas Zimmermann, Peter Weissgerber. *Preprocessing CVS Data for Fine-Grained Analysis*. In Proceedings of the International Workshop on Mining Software Repositories, May 2004.

Appendix A. Sample Edit Action List

The following table presents an initial list of actions that can be undertaken on elements of a UML class diagram, and potential amplifications that can be applied to them. For amplifications that can be applied with various scopes, the suffix “in D/P/O” means “in the current diagram, in all diagrams in the project, in all diagrams in the organization”. The user would select the desired scope when amplifying the action. Note that if a target fulfills multiple criteria, all the corresponding amplifiers would be applicable.

Target	Action	Amplifiers
attribute	show/hide visibility, type, multiplicity	all attributes in class, in D/P/O
	delete	all attributes in class; all attributes with this name in diagram; all attributes with this type in D/P/O
	change to association	all attributes with this name in diagram; all attributes with this type in D/P/O
operation	delete	all operations in class; all operations with this name in class, in D/P/O; all operations with this signature in D/P/O
	show/hide visibility, types, parameter names	all operations in class, in D/P/O
	treat as accessor (convert to attribute, r/o or w/o)	all operations with this name in D/P/O; all operations with this signature in D/P/O
two operations	treat as accessors (convert to r/w attribute)	all operations with this name in D/P/O; all operations with this signature in D/P/O
association	delete	all associations of this kind (e.g. generalization, implementation, relation, dependency) in D; all associations with this name in D/P/O; all associations

		originating from this element in D/P/O; all associations targeting this element in D/P/O
navigable association	change to attribute(s)	
association name	delete	all association names in D/P/O; all association names on associations that have at least one role name in D/P/O
association endpoint	toggle navigability	
association endpoint role	delete	all roles in D/P/O; all roles with this name in D/P/O; all roles for associations with this name in D/P/O
	show/hide visibility	all roles in D/P/O
association endpoint multiplicity	delete	all multiplicities in D/P/O
composition or aggregation endpoint multiplicity	delete	all composition or aggregation multiplicities in D/P/O
two associations between a pair of elements, going in opposite directions	merge	
bi-directional association	split	all bidirectional associations in D/P/O
stereotyped association	delete	all associations with this stereotype in D/P/O
class	delete (all associations with the class are deleted as well)	in project, in organization (?); all classes with same prefix/suffix in D/P/O (when multiple deleted classes have matching names)
	toggle datatype stereotype (changes associations of this type to attributes and vice-versa)	in project, in organization
	mark as collection (class disappears, associations to it are retargeted at "unknown" and gain high	(automatically applies to organization)

	multiplicity)	
	mark as map	(automatically applies to organization)
interface	toggle lollipop notation	this interface in P/O; all interfaces in D/P/O
nested element (package in package, class in package, class in class)	pull inside/push outside	all elements in same container; this element in project, organization; all elements of this kind in project, organization
container element (package or class with nested elements)	take all inside	this element in project, organization; all elements of this kind in project, organization
	push all outside	this element in project, organization; all elements of this kind in project, organization
stereotype	show/hide	this stereotype in D/P/O; all stereotypes in D/P/O
diagram	layout from scratch	
	relayout connectors	
	revert local changes	
	submit: <i>update</i> (execute requested server-side actions and send again), <i>accept</i> (with changes), <i>reject</i> (potentially useful, but don't feel like fixing), <i>destroy</i> (not useful, don't draw this any more)	
two or more classes and/or packages	split into a separate diagram	
any element	show affecting actions (allow "deleted" parts to be recovered, or exceptions to global rules to be created)	
any number of elements	request layout from scratch (these elements and their connectors only)	