

TREND GUIDE

Embedded Software Redesign



MICROCONSULT

TRAINING. COACHING. ENGINEERING.



Team Spirit and Insight:

**Software Redesign with
ARTiSAN Studio and UML 2**

www.artisansw.com

Content

Embedded Software Redesign

Preamble	3
Part I The Decision	5
Part II Reverse Engineering: Cutting the Gordian Knot	12
Part III Refactoring: Make Do and Mend	18
Part IV Reengineering: The Journey is the Reward	25
Info Pool: Recommended Books	32
Info Pool: Tools and Web Tips	33
MicroConsult Services: Training, Coaching, Engineering	34
The Authors	39
Imprint	40

Preamble



Dear Readers,

Take some time to read this Trend Guide – it will be worth it! Enjoy a diverting report with useful information and tips. Maybe you'll even recognize your own situation in some of the passages.

I'd like to thank my colleagues Sabine Pagler, Thomas Batt and Frank Listing for their commitment, expertise and ideas, and of course the managing partners of MicroConsult who gave us support and space to write this Trend Guide.

I appreciate all the valuable suggestions and ideas elaborated by Martina Hafner and Hans Wiesböck from Elektronikpraxis in the context of our common project "Embedded Software Engineering Report".

Special thanks go to Christiane Kapteina from ARTiSAN Software Tools: She convinced her company to support this Trend Guide as exclusive sponsor.

Last but not least, I'd like to thank our readers. Countless downloads and positive feedback have encouraged us to hit the keys once more. We are looking forward to more feedback and suggestions which you are welcome to mail to trendguide@microconsult.de.

Best Regards,

A handwritten signature in black ink that reads "Peter Siwon". The signature is fluid and cursive, with a long horizontal stroke at the end.

Peter Siwon

p.siwon@microconsult.com



Dear Readers,

Software is like a living organism. It thrives under the principle of defensive programming and is processed with minimally invasive methods whenever possible.

Thus, the software architecture and structure often become unrecognizable. At the same time, requirements are getting increasingly complex and development cycles are shrinking. Sooner or later, the next modification or extension crashes the software, and it's high time for software redesign.

If only there was a construction plan or model for us to get at least an idea of the initial structure without having to dig into a million lines of code!

Many modern systems are already based on object-oriented design and developed and maintained with model-based techniques. The current construction plan is constantly available and supports systematic and selective modifications.

Modeling can also help to revitalize older systems. Read all about anti-aging for your software in this Trend Guide.

Enjoy reading – and modeling your software afterwards!

Best Regards,

A handwritten signature in black ink that reads "C. Kapteina".

Christiane Kapteina

Christiane.Kapteina@artisansw.com

The 3 R of Embedded Software Redesign

Part I: Decisions

In 2006, MicroConsult organized several events on **Software Redesign for embedded developers** which were attended by numerous engineers and managers. Feedback proves that a lot of companies regard software redesign as a burning issue.

Software has been growing for decades under frame conditions like real-time, memory optimization, cost optimization and project pressure and achieved a substantial degree of complexity. By now, development teams have reached the limits of their familiar methods and procedures and are desperate to find a solution.

Our Trend Guide provides valuable support to this end.





Figure 1: MicroJones polishes M2C2 after successful Redesign

Credits, Debts and Interest on Software Quality Accounts

The software dilemma is described best using the following analogy. Every development team has three quality accounts:

- the documentation account
- the architecture /source code account
- the process account

If all processes were carried out according to plan, these accounts would show a nice credit balance. Documentation makes orientation in architecture and source code easy, and new team members are trained quickly.

Software architecture and source code are extended comfortably at short notice. The process runs smoothly, from the requirements stage to start-up in the field.

Typical Situation: Software Quality Accounts

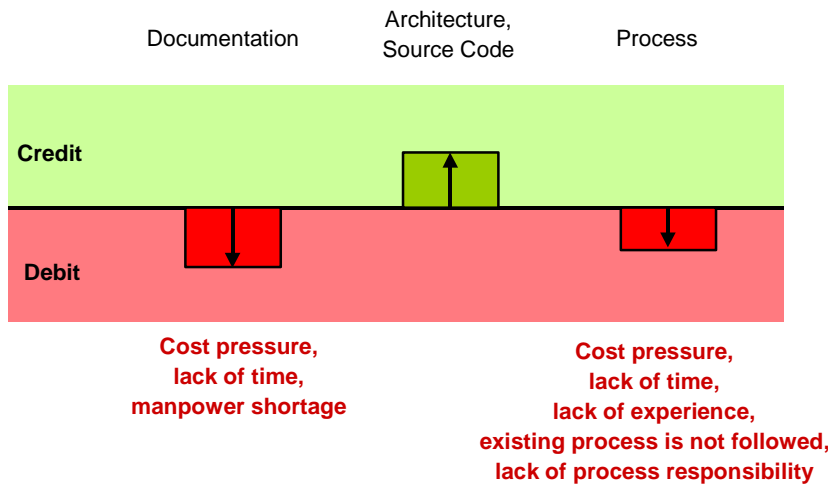


Figure 2: Software quality accounts

Fabulous! And that’s exactly what it is – a fable that is often told but hardly ever comes true. Reality is different and often more like a thriller: A few exceptionally gifted “freaks” build a prototype. The software developer will have plenty of time to program everything “properly” when a customer turns up. But all of a sudden, the customer wants more, with different features, and quickly. The prototype is hastily spiced up to be a product, and additional functions are implemented. The team is well established and documentation is mainly saved in their heads. The architecture is not perfect yet but has some room left for more functionality. The process is a virtuoso juggle with files and tools.

The credit balance of the quality accounts is meager. Growing complexity, loss of knowledge bearers or an expansion of the team soon necessitate loans in order to withstand project pressure. In spite of all good intentions, documentation is not updated even though functionality has grown substantially. The architecture can only be extended with tricks, and after a little while, the process resembles a house of cards that might fall down at the slightest whiff of air. Software developers are forced to hastily plug holes instead of following a systematic process.

On top of that, interest has accrued. Soon the team loses control of the software. Stabilizing the software requires more and more stunts and emergency measures. Old stagers are snowed under with work and newcomers are eager but completely in the dark. Management puts even more pressure on the development team by cracking the outsourcing whip.

In our scenario, the interest is called loss of time, quality, customers, money and motivation.



Figure 3: Dangerous credits

What does your account balance look like?

As far as our experience goes, quality accounts are often emptied much faster than expected. Don't let it happen – you had better not run into debt at all. Suitable “debt redemption” measures are:

- **Reverse Engineering** to improve documentation
- **Refactoring** to provide for an upgradeable software architecture and extendable source code
- **Reengineering** to achieve the required process adjustments

In Short: The 3 R of Software Redesign.

Intensive Care or Pathology?

Before returning to the straight and narrow and become debt-free again thanks to the 3 R, ask yourself these questions: Is the effort worth it? Grim as it may seem, find out if you:

- can keep your software alive,
- want to reuse suitable software sections only,
- dissect a “software body” to learn for the future, or
- bury it all right away and wipe the slate clean.

Probably, some team members are already worried about the problems and assume that things have to change. However, a decision should be based on facts to make the situation tangible. This way, you also take on board those people who are less sensitive – especially management who tends to have a more distant and economic view of the situation.

You should be aware of the fact that filling quality accounts is an investment at the expense of other accounts, such as productivity or profit. However, the interest earned on quality accounts will result in shorter project times and lower costs and amortize the investment soon.

The Diagnosis

Welcome to the medical ward of Software Redesign. The state of our software patient can be diagnosed with three criteria:

- the outer quality of the software
- the inner quality of the software
- project specific requirements

The **outer quality** is assessed by external characteristics. Suitable questions are:

- What operational states or changes result in an accumulation of errors?
- How many error messages appear during system test?
- How many runtime errors occur after delivery?
- How extensive is the functionality?
- Have the above-mentioned numbers changed, compared with the latest software revisions?
- Is there a relation between extended functionality and the effort required to run the system in the field without any problems?
- Were all safety, reliability and performance requirements fulfilled?
- Is the customer satisfied with usability?

You might find out that the implementation of new functionality results in an abrupt rise of errors which are hard to control and which also occur outside of this functionality. Critical dependencies within the software could cause these problems.

It is also helpful to analyze the length of the various project phases. When system tests cannot be carried out within reasonable time, when new products exhibit an increase in childhood diseases or when requirements are knocked on the head all the time, these are clear signs of weaknesses in your process. Error and failure statistics from the field or internal error lists from the development department can provide valuable information.

You should not look for possible mistakes made in the development phase. Find out whether the frame conditions of the development process promote such mistakes and eliminate the causes.

If a system is repeatedly overloaded, this could be due to insufficient memory or CPU performance. Common embedded software engineering tools (linkers, debuggers, emulators) can be used to determine the memory and CPU load.

The assessment of the **inner quality** refers to the software itself. Useful criteria are:

- number of lines of comment,
- available documentation,
- architectural features like portability, maintainability, extensibility, reusability,
- complexity.

New team members or experts from outside the development team are also a source of valuable information. Another useful indicator is the time a new team member takes to make modifications, as well as the “annoyance factor”: how often does the newcomer have to turn to experienced engineers because documentation does not provide sufficient support? If modifications take much longer than before and new engineers require more and more support to this end, it is high time for action.

Tools like CNCC or LocMetrics support a rough evaluation by determining the relationship between comment lines and lines of code (see Info Pool: Tools and Web Tips). However, the results should be verified carefully because the counted lines of comment also contain commented out code, useless comments or mere structuring aids, such as asterisk lines.

Thus, you have to check the listings at least randomly. Comments are often nothing but commented out lines of code from previous programming. There are also comments like “It’s magic” (which basically means “I have no idea why but it works”) or those to remind newcomers of what IF or FOR mean. The joke of the day or statements from a frustrated developer (“Who the h... wrote this crap?”) are also to be found now and then and might be funny but not really helpful. We all know that the number of lines of comment does not necessarily indicate that documentation is helpful and understandable.

The **inner set-up** can also be assessed by means of sample reviews: Does the software contain monster functions or monster code the analysis of which gives you a headache? What about indentation or other structuring aids? How intensely are the program sections interlaced? According to old stagers, code must have a certain degree of aesthetic and transparency to be well readable.

A measure for complexity is the cyclomatic number. The name itself is awesome, but what does it mean? A certain Mr. McCabe agonized over this issue and summarized his findings in a formula:

$$V(G) = e - n + 2p$$

e: number of edges of the graph

n: number of nodes of the graph

p: number of connected components

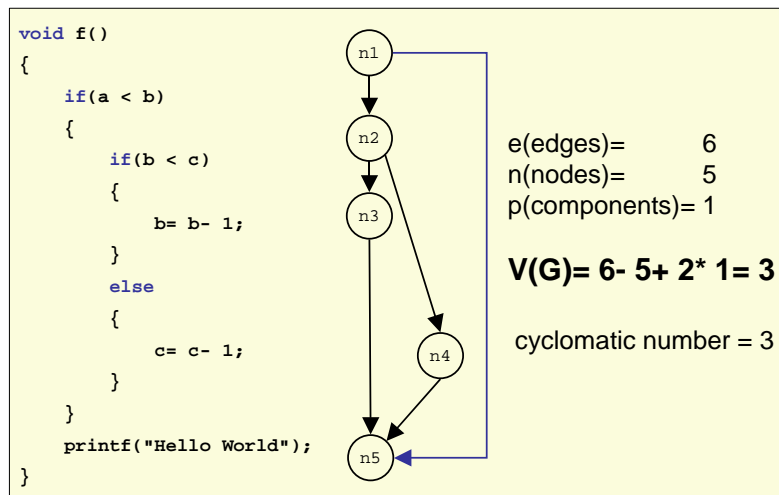


Figure 4: Explanation of the cyclomatic number

This formula should be applied with reservation. It offers a wide range of interpretation, because a high level of complexity on program package level does not necessarily indicate unreadable or insufficiently structured code. Increasing the number of modules or subprograms might in fact enhance understandability. However, a high cyclomatic number on functional level in any case indicates that you should examine your code thoroughly. The cyclomatic number is determined with tools like CCCC-C or CCCC-C++ (see Info Pool: Tools and Web Tips).

Moreover, the **requirements of your projects** are efficient decision aids. If changes are planned in any case – selecting a new processor, taking workload off the team by means of outsourcing, or extending functionality – why not use the opportunity and make a clean sweep right away? Outsourcing can be a sensitive issue. A time or cost problem has caused deficits in your documentation, architecture or process and made your management consider outsourcing. Suitable outsourcing measures, however, necessitate an update of the documentation and a reformation of the process – which might eliminate the cause of the previously mentioned time and cost problems.

The simplest motivation for change is legal regulations, customer requirements or an innovative competitor who has already decided to reform his software development process. Maturity levels according to CMMI or SPICE, or the demand for standards – UML (Unified Modeling Language), MISRA (The Motor Industry Software Reliability Association) etc. – might also initiate a reformation process.

When using tools, you cannot do without an evaluation of code and documentation in order to properly validate the results achieved with these tools.

The quality-related effort for Reverse Engineering and Refactoring can be estimated as follows:

The lower the level of abstraction, the more probable it is that code, data and hardware platform are highly interlaced, and this of course increases the Reverse Engineering effort. A good reason to consider object-oriented and/or model-based solutions for the next software generation.

For statements on the quantity-related effort, you should assess the following steps for analysis and documentation. This can be handled within the team with the help of an experienced project coach and usually does not take longer than one or two days. To this end, representative software sections are selected and evaluated exemplarily. The effort is projected up to the overall software. Evaluation refers to process steps like preparation, follow-up, analysis of architecture, file and function headers, code, etc. The effort for code documentation, for example, can be estimated based on the experience gathered from Fagan inspection. Insufficiently documented code makes it hard to implement an architecture analysis without code analysis.

Consequently, the effort mainly depends on how often and how deep you have to look inside your code in order to comprehend its internal coherence. This evaluation can be further refined in the course of Reverse Engineering.

A realistic estimation of available resources for Redesign is another important basis for decision-making. The 100% solution is quite rare, but there are lots of acceptable compromises for restructuring, recoding and process adjustment. For example, you can start with focusing only on those packages which involve high risk or need to be handled in any case. And it is up to you to decide how fine-grained the measures should be in order to bring light into the dark again.

These decisions require common sense and professional instinct. Tools can help, but the team decides and management should support the decision. Experienced project coaches are very helpful in this process. They bring in experience from software redesign projects with completely different frame conditions. In addition, coaches take the position of (largely) independent and impartial observers and moderators. They are not routine-blinded or subject to strong emotional pressure because they are not personally concerned with your project.

Conclusion – useful procedures for Redesign:

1. Find objective arguments for a Software Redesign to facilitate an economic assessment of effort and benefit.
2. Evaluate the outer and inner quality of your software and the frame conditions of your software projects. Find pro and con arguments.
3. Make use of tools but rely on your own know-how.
4. Don't strive for perfection, go for the feasible solution.
5. Involve project coaches.
6. In the end, common sense and professional instinct turn the balance.

The 3 R of Embedded Software Redesign

Part II Reverse Engineering: Cutting the Gordian Knot

After analyzing the architecture, source code and documentation and estimating the effort, the team members and management have reached the following conclusion:

A better documented knowledge of the software is urgently required in order to maintain its value and to use it as basis for further development. A new development is more expensive than reusing existing code.

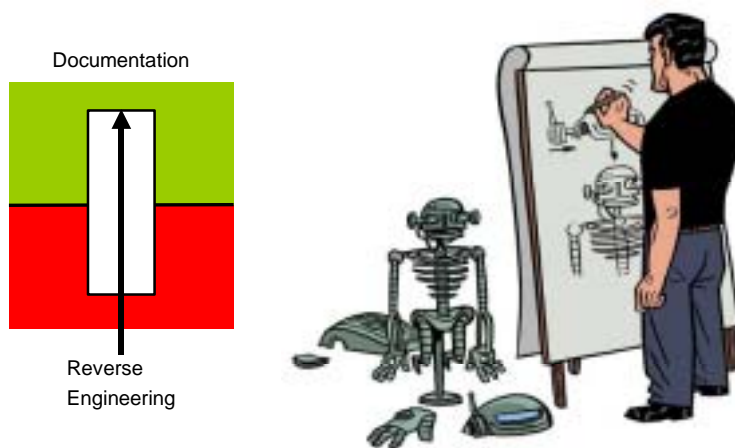


Figure 5: Reverse Engineering fills the documentation account

The method applied now is called **Reverse Engineering**. In extreme cases, all you have got is code from which you extract and document information on architecture, concept, principles, etc. This task can be compared to untangling a heavily knotted rope instead of simply taking a sword and cutting the Gordian knot like Alexander the Great. The problem is that you need much more information than is actually found in the code. But how do you get it?

This is what you probably know about your source code:

- There is not enough information on its internal coherence (architecture and structure).
- There are sections that nobody dares to touch without getting a pain in the stomach.
- It is not clear which requirements were in fact implemented and which weren't.
- There is "zombie code": dead code that keeps vegetating in the memory.
- It is not clear which sections can be reused and which need to be reprogrammed.

With **Reverse Engineering**, you can awaken to this nightmare and then get rid of it. The more you know about your software, the faster you can make it better.

You have probably suspected that some effort is required to this end, as we already found out in Part 1. Unfortunately (or luckily), there are no tools yet that are intelligent enough to extract the motives of the coder from the source code. After all, the formula $E=mc^2$ does not indicate Einstein's train of thought on the theory of relativity. Things are usually not that complicated with software. You mainly need to get the following information:

Superior information

- Architecture
- Implemented features

Code documentation

- File headers
- Function headers
- Description of code blocks and lines of code

The procedure basically consists of three steps:

- Analyzing the software
- Deciding what happens to the identified software packages
- Documenting the results

There are two analysis methods: **Architecture analysis** deals with the structural coherence of the software package at module level. **Code analysis** covers formal issues like file headers, function headers and visual structure as well as functions and algorithms – mainly diligent work. It can be quite difficult to elaborate the finesse of functions and algorithms if the software incorporates tricks of real-time and memory optimization offered by an assembler.

The same is true if you want information about the intended architecture. It might have become heavily overgrown in time, like an unattended allotment garden.

Architectural Analysis: Drilling and Digging

Why is it so important to document architecture? Architecture is the key to an understanding of the whole system and its dependencies. You need to know your architecture to be able to define how and where system properties can be modified or extended without putting the system at risk – like the construction plan for a house showing the locations of supporting walls, pipes or supply lines. In addition, the architecture supplies valuable information on the weaknesses and limitations of the system. This information is not fully available in the code, and many companies even lose it when the knowledge bearer quits.

Imagine you have to reconstruct the organization chart of a company by asking all employees about their functions and communication paths. Reconstructing architecture from existing code is a similar task. Reconstruction might be a logical measure but does not necessarily have to conform to the intended purpose.

Consequently, the system architect should not only have enough time to get things running but also time to document the mechanisms. By the way, good system architects are in high demand! You certainly don't want them to carry their know-how to other companies and leave your company behind and in the dark.

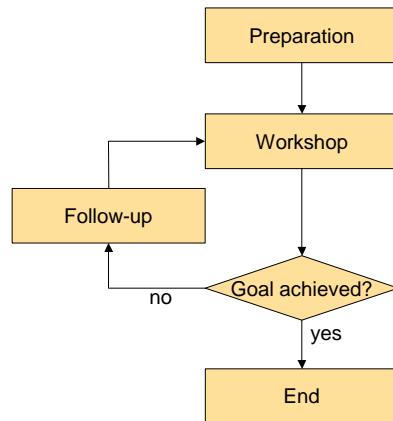


Figure 6: Flow diagram – Architecture analysis

Probably the most crucial and most difficult part of preparation is finding a meeting date that is suitable for all knowledge bearers and engineers involved. The effect (lack of time) often hinders the correction of the cause (missing documentation) – a vicious circle.

However, Reverse Engineering in the figurative sense means washing, scrubbing AND getting wet – with the prospect of having everything clean afterwards. People who never have a proper wash should not wonder why they smell bad. By the way, we'll get back to the "smell" issue later in this Trend Guide.

There comes a time when the important must have priority over the urgent. This is an old management rule and you are free to remind your managers of it in due time. The invitation to a workshop must contain information on the goal, agenda and duration. To get initial indicators on the internal coherence of your software, it is helpful to generate reports with tools like Doxygen or Doc-O-Matic (see Info Pool: Tools and Web Tips).

A documentation tool for representing software architecture should be selected with utmost care. The most simple method is a flip chart and colored pencils, but UML tools are of course much more flexible and offer a wide range of useful diagrams (see Info Pool: Tools and Web Tips). In addition, they follow a standard that is more and more prevailing in software engineering. Thus, you provide a sound basis for a future modification of your software architecture.

Finally, the workshop can start. It is recommended to first break down the software into layers and assign them to the software packages identified below. This could look as follows for embedded systems: One layer is the user interface with the package “human Interface”. Another layer follows for all control functions, with packages like “drive control”. Finally, there is the hardware interface layer with the packages “sensor and actuator driver” and “communication driver”.

The following figure shows such a rudimental architectural model in a UML diagram.

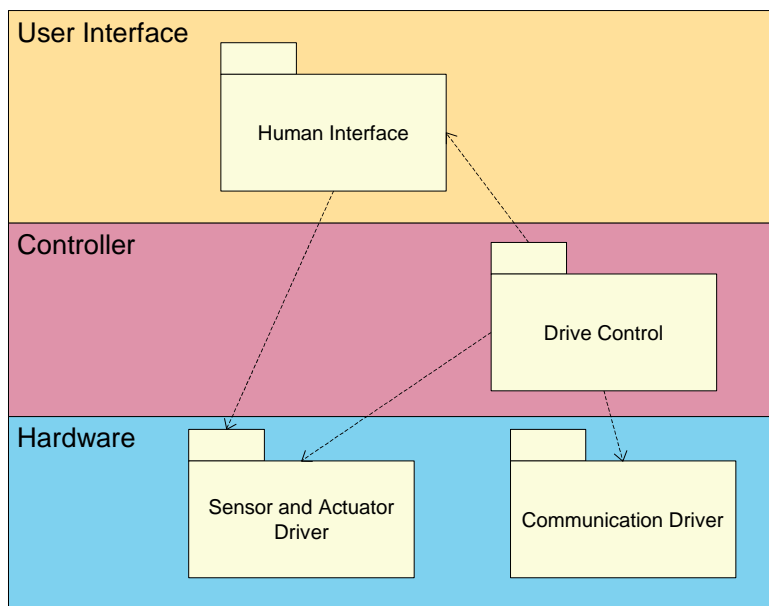


Figure 7: Rudimental architectural model

In addition to the packages, the model also shows dependencies by means of dashed arrows, each pointing towards the dependency. The controller obviously depends on both the user and the hardware interface.

This simple example shows how useful these diagrams are. Highly complex coherences are clearly visualized. You should define the extent to which UML shall be used also in the further proceedings and which functionality the tool shall offer (code generation, repository). Then select a tool that you may have in your company, or look for another suitable tool. Depending on your requirements and budget, you can choose between less convenient drawing tools (Visio) and comprehensive modeling toolsets (ARTISAN, Rhapsody, etc.).

In this phase as well, MicroConsult project coaches offer support and many years of application experience in the embedded environment. At least one team member should have a good knowledge of the use of the method and tool (see Info Pool: Training).

The workshops could be managed by an independent moderator or project coach who keeps an eye on the progress and goal. An independent recorder is also recommended so that all others can focus on professional aspects.

Prepared reports enhance orientation. If required, the source code is inspected in order to safeguard the coherences. In the course of the meeting, tasks and related deadlines are defined. Team members who did not participate can thus be involved. The most important rule is:

All findings must be documented AT ONCE so that nothing gets lost.

At the end of the workshop, verify if the defined goals were achieved and settle a date for another workshop if required.

One useful side effect of the workshop is that it is probably the first time that the complete team gets an idea of the architecture.

Code Analysis: Sieving and Sorting

A closer look into the software, especially in case of Assembler and C code, shows that it is largely "handcrafts" which can be supported by tools only rudimentally. Information on Fagan inspection is helpful for an assessment of the required effort. For source code analysis, a speed of approximately 100 lines of code per hour is assumed.

It is obvious that this task cannot be handled in a rush, alongside everyday work. The project schedule has to consider the time required to this end.

Workshops, small teams and coordinated individual measures are recommended for implementation. We advocate a kick-off meeting to define the goals and the scope of code analysis, in connection with a time schedule, milestones and responsibilities. The results should be added to the project plan. In addition, templates for file and function headers or other formal conventions are agreed upon during the preparation phase.

A systematic and step-wise approach is crucial for implementation, with documentation being structured from high level to detailed software features. For example, you first update the file headers and then continue with the function headers. If required and possible within the defined time frame, structure the source code clearly and make comments where necessary. The findings also have to be documented at once.

The following tools offer suitable support:

- The freeware tool Doxygen (see Info Pool: Tools and Web Tips) extracts information previously marked with tags. It describes the dependencies of code and file headers and thus represents the call hierarchy of the functions. Suitably prepared code can be transferred conveniently into compact documentation.
- The commercial tool Doc-O-Matic offers similar functionality but also documents function headers that do not contain special tags, according to the supplier. This is achieved by an intelligent analysis of the comments. The results should be reviewed carefully.

Doubles and Copy Gremlins

Duplicated code occurs frequently in programs generated with the copy & paste method. It not only causes redundancies but is also a source of error. These errors are probably overlooked at a later stage or not corrected consistently across all copies. Tools like *PMD*, *duplo* or *same* are especially helpful when it comes to eliminating such weaknesses later in the Refactoring phase.

Conclusion – useful procedures for Reverse Engineering:

- Reverse Engineering is an element of project planning.
- The effort can be estimated using the guide values from Fagan code inspection.
- Scope and form of the documentation depend on the goals and available time.
- Systematic procedures are a must – from the outside to the inside, from the important to the noncritical.
- Some tools offer orientation or decision aids and simplify the future use and extension of the documentation.

The 3 R of Embedded Software Redesign

Part III Refactoring: Make Do and Mend

The process of Reverse Engineering provides for architecture and source code optimization in many ways: It documents of the status quo, it identifies suitable tools, and it gives the team a clear idea of the structure and behavior of the software.

Strictly speaking, Refactoring means enhancing the internal software structure and code without changing the external behavior of the software or the system.

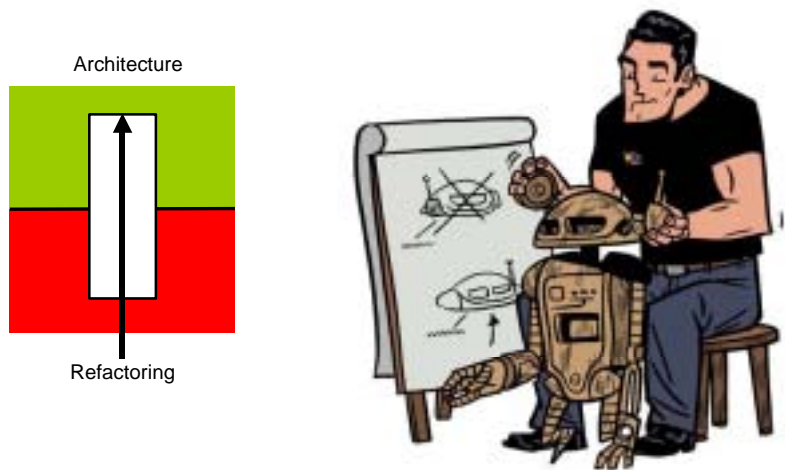


Figure 8: Old functions in improved design

The following preconditions have to be met:

You must be able to prove that the modified system can execute all existing functions of the original system without taking along the now known weaknesses.

Ideally, there is an automatic black box test that can also be applied to the new software. There should be a suitable test, in any case, to validate the correct implementation.

Preconditions for Successful Refactoring:

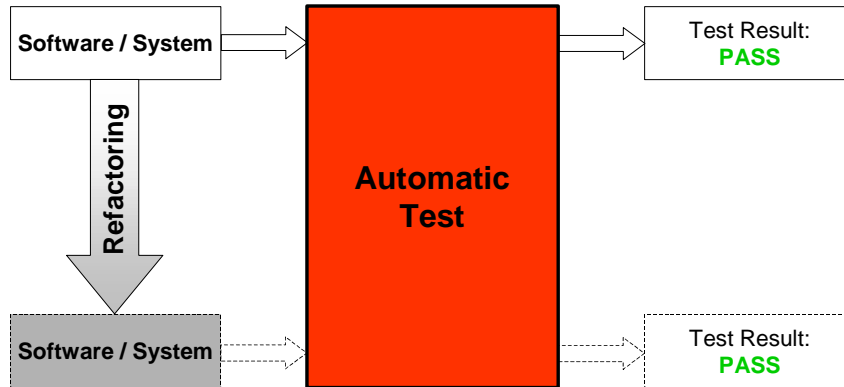


Figure 9: Refactoring and test

Refactoring might be required for different reasons. One of the most common ones is the method of moving code from the biological to the electronic memory, i.e. from the brain to the computer, which might work for simple systems. In more complex systems, however, this method is sure to cause chaos quickly. Another cause of obscurity the generation of serial software from prototypes. In most cases, the good intention to reprogram the software properly once a customer has been found is sacrificed to time pressure. "After all, it works. We're still (!) in control of the situation. When there's more time, we will catch up on everything." A seemingly good argument for omitting documentation.

Does this sound familiar to you? Long-time developers may be virtuoso programmers, but many of them seem to ignore the creeping danger. Or they leave the company in order to make it right from the start somewhere else and escape the sword of Damocles – a difficult situation for their successors.

Maybe you have made everything right so far. However, tasks and environments are getting more and more complex. It is difficult to identify the right time: When is it more laborious to go on with well tried processes instead of taking courage to wipe the slate clean? In most cases, it is not a matter of recognition but rather the willingness to face reality – in other words: routine-blindness.

A good remedy: Lean back with your team at least once per year and take a look at your work and your methods. External coaches can help in this phase by initiating and encouraging such an "awareness process".

There are a lot more good opportunities for Refactoring – situations in which you have to take the bull by the horns anyway:

The integration of new functions, the elimination of errors, a certification making changes inevitable, a change of the software development paradigm towards OOP, the implementation of a real-time operating system, the integration of other external software products, software porting to new hardware platforms or porting to multicore platforms which requires your software architecture to be highly flexible. The simplest motivation: a customer asks for this application, or competition already offers solutions to this end. You have to be able to build a software architecture that can be ported to various platforms and extended to meet future requirements if you want to benefit from innovation cycles in the long run.

Before jumping in to architecture and code, you have to agree on the Refactoring goals:

- Improved maintainability
- Improved expandability
- Better understandability
- Enhanced time behavior
- Reduced memory usage
- Higher reliability
- Higher safety
- Platform independence
- Shorter project times
- Shorter training times
- etc.

This list makes it obvious that the prime goal of Refactoring is to optimize **several** goals instead of maximizing one goal only. In any case, excessive platform-dependent code optimization aiming at a minimization of memory requirements or runtime should be considered critically when it results in inflexible or hardly comprehensible results, because this increases the risk of error.

(A short, wicked question addressed to the automotive industry. The main memory size of control units was reduced in order to cut unit costs. Has this measure in fact yielded a revenue that is higher than the resulting additional development costs? Did this approach also consider that extensive handcrafts would be required, making cost-intensive product recalls more likely?).

Hence, there are plenty of reasons and goals.

For the realization of our goals, we differentiate between big and small Refactoring. Big Refactoring deals with the architecture beyond functions, data and classes. It aims at introducing an architecture in case there isn't one yet, or improving architecture in order to realize the defined goals on architectural level. **Small Refactoring** refers to the source code or class level. Let's start with this approach.

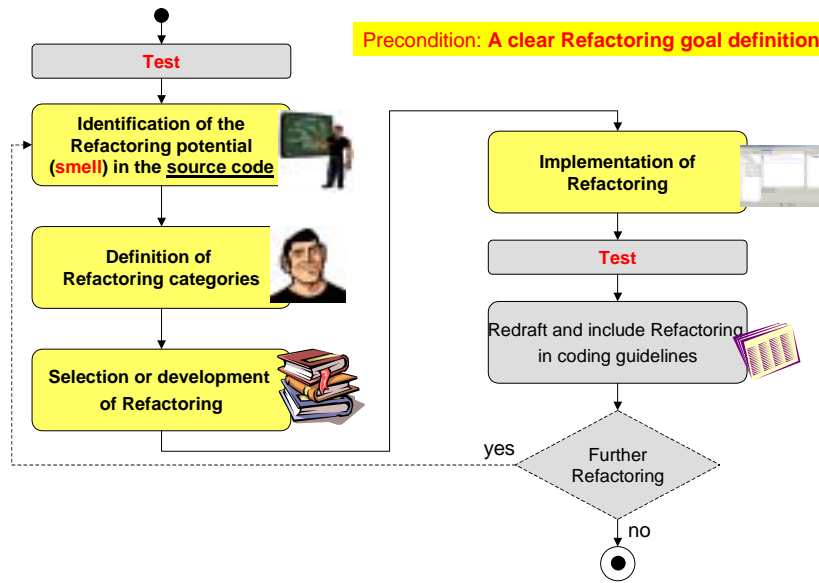


Figure 10: Flow diagram – small Refactoring

Run a specially created or existing test to confirm the behavior of the available software. Then define the Refactoring potential. The findings from Reverse Engineering are a good basis for this. The English speaking developer refers to “bad smell” in this context, in other words: “What parts of our software smell bad?” Then you define the goal of the Refactoring process and the Refactoring categories to be used. Category refers to a description of the basic measures to achieve the goal. Each Refactoring category describes several possible Refactoring methods. Refactoring itself describes the concrete implementation based on source code.

Example¹:

Definition of the goal:	Source Code Optimization Enhanced reusability
Refactoring category:	Composing Method This category lists the Refactorings for the recoding of functions.
Refactoring method:	Extract Method Functions can be recoded by generating an additional function and calling it from within the existing function.
Refactoring category:	Simplifying Conditional Expressions This category lists the Refactorings to simplify conditional expressions.
Refactoring method:	Decompose Conditional Conditional expressions can be simplified by executing the conditional inquiry and all possible branchings through a function call.

¹ Refactoring: Improve the Design of Existing Code, Martin Fowler, Addison-Wesley, ISBN 0-201-48567-2

Many known Refactorings can be run automatically with editors or a plug-in (see Info Pool: Tools and Web Tips). After Refactoring, a test verifies if the behavior of the software was maintained or if the desired behavior was achieved.

Some more tips:

- Structure your software as clearly as possible.
- Find meaningful names for variables, procedures, classes and methods to make your software understandable even without comments.
- Do not only stick to existing Refactoring catalogs but develop your own ones.
- Adjust the programming guidelines to the results.

Big Refactoring is basically the same. However, an incremental method is applied for complex software, with the architectural level being handled first.

This process is usually followed by small Refactoring. Thus, we basically have a combination of small and big Refactoring. The findings also have to be documented in new architecture and coding guidelines.

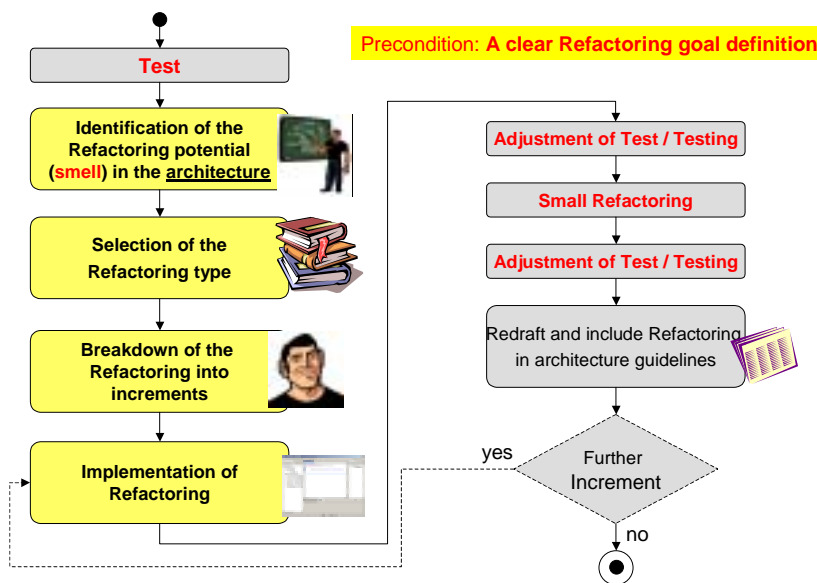


Figure 11: Flow diagram – big Refactoring

Example:

Reusability, maintainability and extensibility shall be enhanced. Object-oriented modeling with UML and subsequent coding in C could be a suitable Refactoring method to achieve this goal.

The starting situation is the following counter module, procedurally coded in C:

```

int count;
int min;
int max;

void Counter_init(int par_count, int par_min, int par_max)
{
    count = par_count;
    min = par_min;
    max = par_max;
    Counter_print();
}

void Counter_counting(void)
{
    if (count >= min && count < max)
    {
        count = count + 1;
        Counter_print();
    }
    else
        count = 0;
}

void Counter_print(void)
{
    printf("Min value = %i ", min);
    printf("Max value = %i ", max);
    printf("Count value = %i\n", count);
}
  
```

With an object-oriented method, you get this UML class diagram:

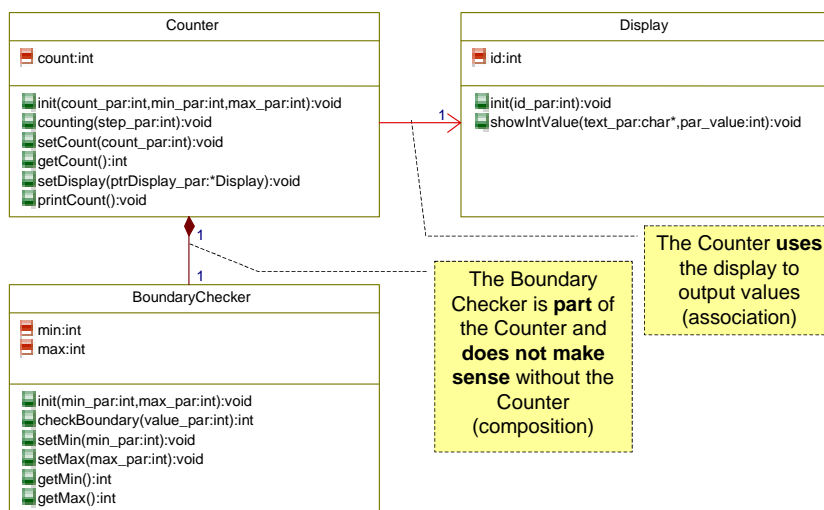


Figure 12: Class diagram for the counter

If you keep the software model and related source code consistent, the UML description also supplies the source code documentation. At the same time, the graphic UML description enables new project members to understand the source code. The above-mentioned UML tools can be used here as well (see Info Pool: Tools and Web Tips).

Most development environments for the PC world (e.g. integrated Refactoring) are more powerful than development platforms for microcontrollers. Therefore, PC development environments should be used also for embedded software wherever possible.

Conclusion – useful procedures for Refactoring:

- Refactoring is part of the development process and project planning.
- Refactoring requires good documentation of the existing software.
- Refactoring can be started at the architecture and source code level.
- Every Refactoring process has a goal that needs to be kept in view.
- After Refactoring, the guidelines for coding and architecture might have to be adjusted.
- Only a test before and after makes results comprehensible.

The 3 R of Embedded Software Redesign

Part IV Reengineering: The Journey is the Reward

The journey is the reward. This does not sound like a wisdom from software development. Meaningful sayings, however, are ageless and independent of technology. The substantial effort for Reverse Engineering and Refactoring is much more due to the journey than to the reward.

In our context, the goal is executable and fault-free software. This goal is achieved much faster and easier by future Reengineering of the software development process. Remember that once a project is finalized, the next one is just around the corner.

According to our analogy, debts incurred in a project usually have to be paid back along with the interest in the next project (at the latest). Even if you are seriously tempted, faster does not necessarily mean better, especially when you want to provide a basis for future product generations.

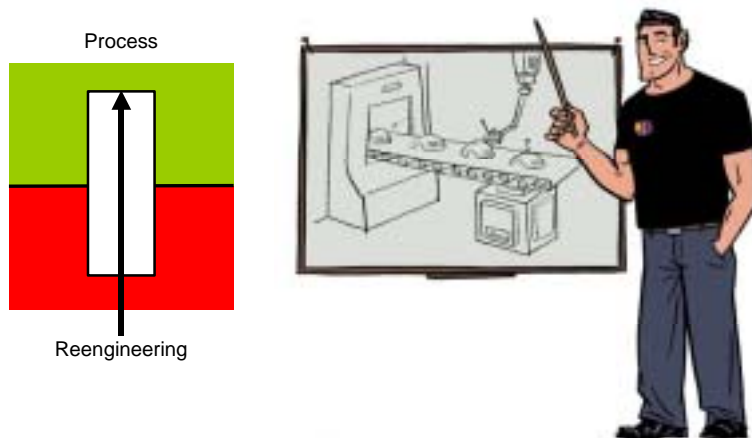


Figure 13: Process improvement

Figure 14 shows typical elements of a process. It consists of so-called core workflows or core tasks, such as requirements analysis or implementation that accompany the process from the idea to the finished product. The progress is divided into phases timed by milestones.

This way, you get defined synchronization points and intermediate results facilitating an evaluation of the project progress and clarification of further courses.

This representation is more realistic than the waterfall model or V-model because it shows each workflow accompanying the whole project, albeit with different intensity.

For example, testing takes place not only at the end of the implementation phase but repeatedly at intermediate stages and thus delivers preliminary results.

In addition to core workflows, there are supporting workflows, such as the provision of infrastructure or project management. They don't have a direct impact on product development but facilitate a smooth project flow.

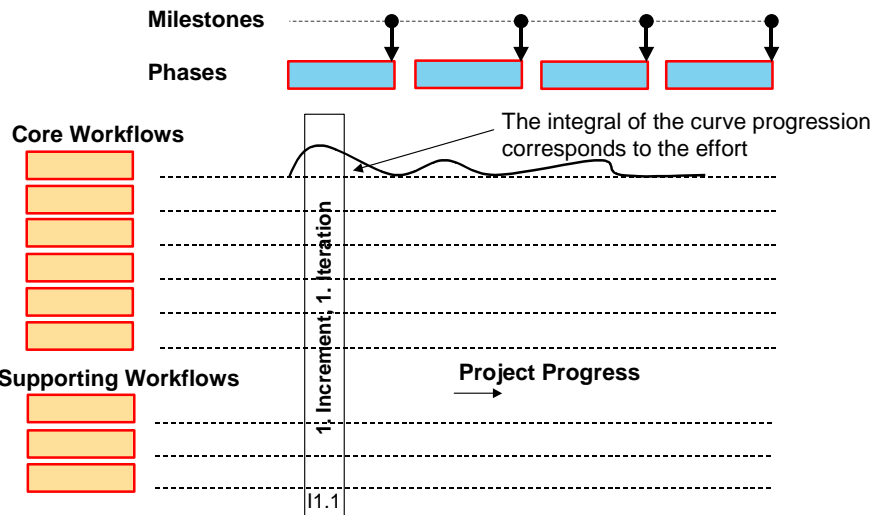


Figure 14: General process scheme

Modern process models support incremental and iterative procedures. Incremental means that the overall project is itemized into subprojects (increments). The increments independently run through the defined workflows until the integration stage is reached. Increments can be created in parallel and/or serially with each other.

Especially in software development, there are only a few methods to get defined statements on executability on the target by means of model simulation. Iterative procedures are useful to this end.

Example: Faulty behavior is identified when an increment is executed on the target. The developer determines which of the previous workflows causes the faulty behavior. He corrects the cause in the respective workflow and runs the increment through all subsequent workflows once more. The first iteration can be followed by further iterations, until the faulty behavior is completely corrected.

Especially in projects with new technologies, tools or methods, iterations across all workflows enable developers to implement valuable experience in further iterations – in each phase of the project.

The workflow itself describes which activities create output artifacts from so-called input artifacts and which roles are involved in this process. An artifact could be hardware, data books, documentation, test protocols, database entries, etc. In the workflow, activities describe how the artifacts are processed.

Method and notation are described for each activity, and there are guidelines for each artifact defining its structure. If the artifact is a document, the structure is described with templates.

Roles could be developers, testers, project managers or purchasers. A role can – but does not have to – be assigned to one person. For example, in many smaller-sized companies, project manager, developer and tester are combined in one person. Combining the roles of tester and developer, however, often entails a role conflict.

Each role is assigned tasks, required abilities and know-how. These definitions are helpful for human resource decisions.

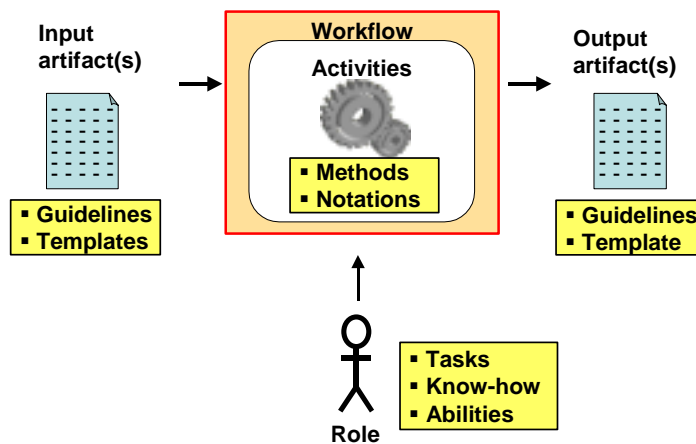


Figure 15: Workflow

All descriptions are summarized in the process documentation. Of course, every developer or development team has got their own process along which they proceed. However, such processes are often highly intuitive (some may even say chaotic) and documented insufficiently or not at all. They may work quite well and bring fast results when development teams are small and system complexity is relatively low. However, if tasks are repeatedly carried out twice or not at all, the limitations of such processes become obvious quickly.

Documentation offers one more advantage: documenting means reflecting, and reflecting means identifying the potential for improvement.

You have to show your colors, at the latest, when the process model is certified – according to ISO (International Organization for Standardization), SPICE (Software Process Improvement and Capability Determination) or CMMI (Capability Maturity Model Integration).

Processes that are also learning processes will change and develop over time. Consequently, you should describe your processes right from the start. This makes it easier for you to identify weaknesses and make changes.

We recommend the use of UML for several reasons:

- UML is an industry standard.
- UML is widely used in many companies.
- UML is supported by case tools.
- The process model is easily adapted when UML is used.
- Process documentation is automatically generated from the UML model in different formats (e.g. Word, PowerPoint and HTML).
- Process documentation is scalable with individual templates.
- The content of the process documentation can be selected from the UML model. Thus, different process layers can be generated for different roles.

MicroConsult has many years of experience with UML and knows the advantages. We have developed a process model for UML based embedded software development that is easily adapted to our customers' requirements.

For this model, we have combined the numerous published processes – V-model, USDP (Unified Software Development Process), RUP (Rational Unified Process), MSF (Microsoft Solution Framework), XP (Extreme Programming), OE (Object Engineering), ROPES (Rapid Object Oriented Process for Embedded Systems), COMET (Concurrent Object Modeling and Architectural Design Method) or Octopus (Object-Oriented Technology for Real-Time Systems) – with our own embedded development experience to form the process model **COPES (Customizable Object Oriented Development Process for Embedded Systems)**.

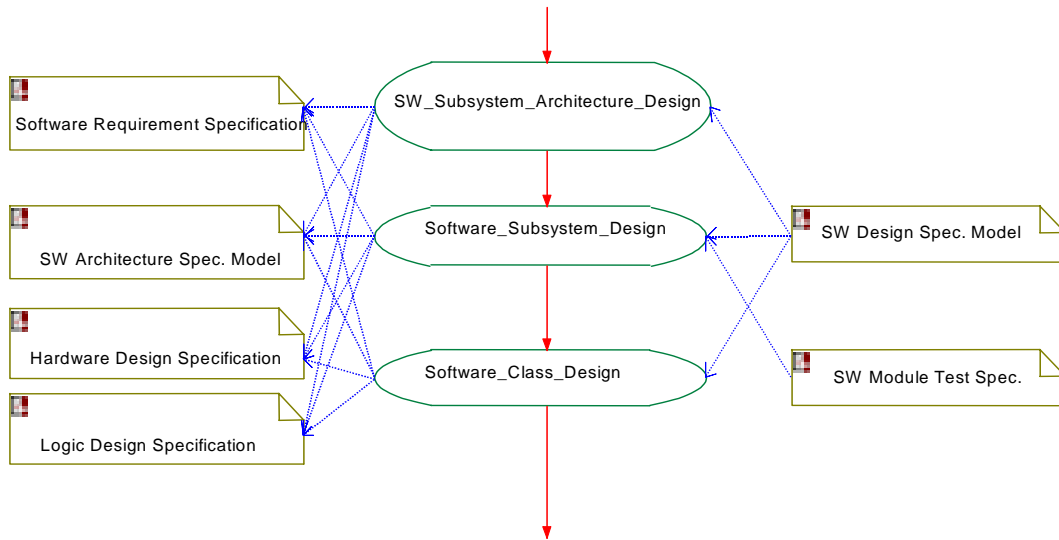


Figure 16: Section of COPES - Software design workflow (UML activity diagram)

Figure 16 shows a section of this model. The input artifacts are on the left and the output artifacts on the right. The work steps are described in the middle. The dashed blue arrows describe the dependencies of the model elements.

The arrows point towards the dependency. The red continuous arrows between the work steps indicate the direction of the procedure.

If UML is used for development, the advantage of this description is that both process and software can be described with the same notation and are moreover available through the same tool.

Based on our experience, we have included **Software Redesign** in our process model COPES.

The following pictures present the defined role assignment. These process models are an adjustable framework that has already proven to work perfectly in practice.

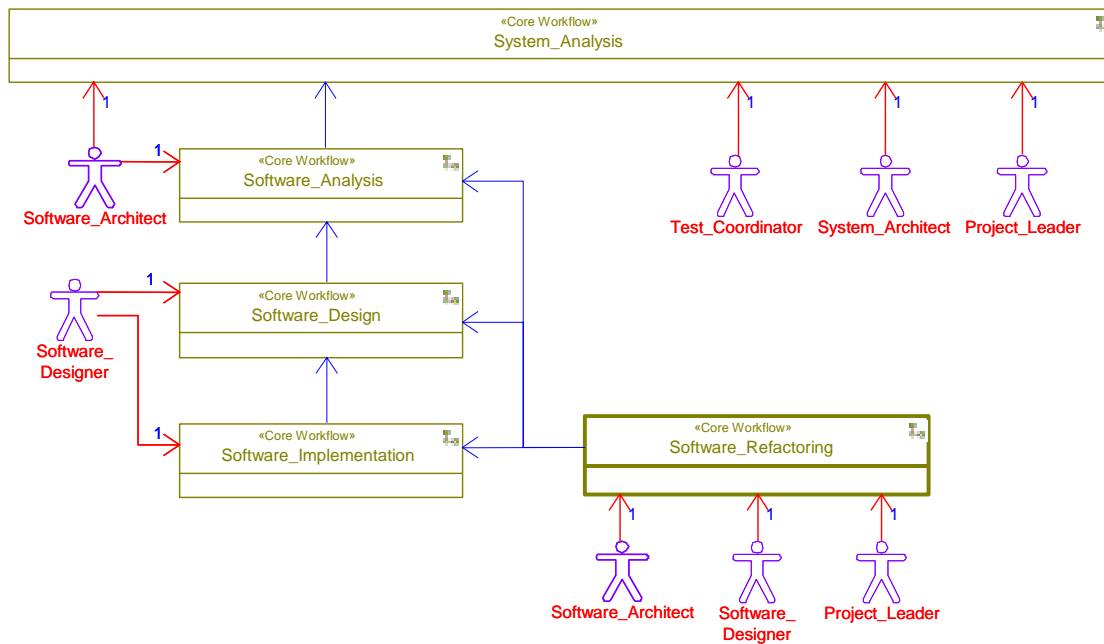


Figure 17: Section of COPES:
Refactoring workflow integration with role description (UML class diagram)

The **process documentation** should be created or adjusted such that it describes all important roles even if one person takes on several roles in the case of smaller teams. This provides the basis for identifying role conflicts or approaches for a more efficient division of work. The process has to be viable and supported by a majority of the team, otherwise it will not work. Thus, consensus is indispensable – especially when achieved through an active exchange of opinions and experience in the determination phase.

With a UML model like COPES and the support of experienced project coaches, this goal is often achieved in a few days only.

Conclusion – useful procedures for Reengineering:

- Process descriptions are helpful and necessary.
- Tool-based process modeling offers a lot of advantages.
- The involved persons should actively participate in the generation of process descriptions.
- Existing models like COPES simplify process definition.
- Project coaches help you avoid detours on your way to a viable process.

Summary:

A regular and critical review of documentation, programming, architecture and process is indispensable for identifying and correcting weaknesses at an early stage of the process. It should happen before fatal loans are taken out on software quality accounts. External support brings substantial benefits. We offer assistance to make Software Redesign a success – with information, training, coaching and engineering services.

Supplement the credit balance of your quality account with MicroConsult.

For successful system development:

Training Coaching Engineering



Figure 18: Credit for your quality account

Info Pool

Recommended Books

Refactoring. Improving the Design of Existing Code

Author: Martin Fowler

Publishing house: Addison-Wesley

ISBN 0-201-48567-2

www.refactoring.com

Refactoring to Patterns

Author: Joshua Kerievsky

Publishing house: Addison-Wesley

ISBN 0-321-21335-1

Refactorings in großen Softwareprojekten: Komplexe Restrukturierung erfolgreich durchführen

Authors: Stefan Rook, Martin Lippert

Publishing house: dpunkt.verlag

ISBN 3-89864-207-0

Die UML-Kurzreferenz 2.0 für die Praxis

Author: Bernd Oestereich

Publishing house: Oldenbourg

ISBN 3-486-57788-3

UML 2 glasklar

Authors: M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins

Publishing house ISBN 3-446-22575-7

Real Time UML (Third Edition)

Author: Bruce Powel Douglass

Publishing house: Addison-Wesley

ISBN 0-321-16076-2

Info Pool

Tools and Web Tips

UML Case Tools:

ARTiSAN Software Tools: ARTiSAN Studio
www.artisansw.com

Telelogic: Rhapsody
www.telelogic.com

Sparx Systems: Enterprise Architect
<http://www.sparxsystems.com/>

Editors and Plug-ins for Refactoring:

Ideat Solutions: Ref++
www.refpp.com

Xref-Tech: Xrefactory for C++
<http://www.xref-tech.com/>

SlickEdit: SlickEdit
www.slickedit.com

Eclipse Organization: Eclipse
www.eclipse.org

Finding Duplicate Code:

Freeware: PMD
<http://pmd.sourceforge.net/>

Freeware: duplo
<http://sourceforge.net/projects/duplo>

Freeware: same
<http://sourceforge.net/projects/same>

Code Documentation:

Freeware: Doxygen
www.doxygen.org

toolsfactory software: Doc-O-Matic
<http://www.toolsfactory.de/>

MicroConsult is not responsible for the contents and functions of the above-mentioned links. The providers of these websites have the sole responsibility.

MicroConsult Services

Related Training – The Optimum Equipment for Various Depths

MicroConsult Training provides the optimum knowledge basis for you to fulfill the requirements of your projects. Our training experts have extensive experience and apply a practice-oriented methodology to make even complex subjects understandable. You are optimally prepared to make efficient use of the right equipment after the training.

For current training dates and the complete program, visit us at www.microconsult.com.

Requirements Engineering and Management for Industrial Developers

Training Objectives:

Understanding and evaluating the requirements process and implementing it in your company; optimizing existing processes.

Prerequisites: None; project experience advantageous.

Content:

Development process; identification of requirements; documentation based on examples and graphic UML tools; acceptance tests; management; practical exercise using a typical development scenario as an example.

Duration: 4 days

UML Basics: Object-Oriented Analysis with UML

Training Objectives:

Competent use of analysis/design methods and the UML representation methods.

Prerequisites: Programming experience, e.g. CHILL, C, Fortran or C++.

Content:

OO terminology; basic object-oriented concepts ; presentation of basic OO concepts using the UML class notation; dynamic behavior of OO software; implementation in programming languages.

Duration: 5 days

MicroConsult Services

Object-Oriented Analysis for Embedded Systems

Training Objectives:

Assessing the use of UML for embedded systems; systematic development of software systems from requirements analysis to design.

Prerequisites: Experience in embedded projects; knowledge as conveyed in our training “UML Basics: Object-Oriented Analysis with UML”.

Content:

UML and embedded systems; development process; requirements analysis; design; practical exercise (e.g. developing an embedded application with UML; following a typical development process).

Duration: 5 days

Design Patterns

Training Objectives:

Efficient use of the most important Design Patterns in embedded development.

Prerequisites:

Experience in an object-oriented language.

Content:

The OO paradigm and UML as basis for Design Patterns; delegation and polymorphism; creational patterns; behavioral patterns; structural patterns; practical examples for the use of Design Patterns in embedded development; interaction of several patterns; overview of architectural patterns.

Duration: 4 days

Software Quality for Embedded Systems

Training Objectives:

Systematic definition of the desired software quality; knowledge of software quality standards; validation of the achieved quality.

Prerequisites:

Software system project experience.

Content:

Development process quality; computer and software quality; validation of product quality; practical exercise, e.g. code test or test result analysis.

Duration: 5 days

MicroConsult Services

Software Project Management

Training Objectives:

Methods and procedures for development projects; optimization of own projects.

Prerequisites: Project experience.

Content:

Project management; project organization; proven development process models; successful project start; realistic planning of product development projects; methods and processes for project controlling; project simulation with SimulTrain®.

Duration: 5 days

Software Test – Structured and Efficient Testing of Embedded Systems

Training Objectives:

Professional testing: planning, evaluating and implementing tests.

Prerequisites:

Basic knowledge of a high level language, e.g. C/C++.

Content:

Testing in the embedded market; software quality for embedded systems, requirements analysis; test specification and planning; static test from review to LINT; integration strategies; dynamic testing; test evaluation; cost-effective testing.

Duration: 5 days

Embedded C++: Object-Oriented Programming of Microcontrollers with EC++

Training Objectives:

Benefit from OO methods for embedded system development and avoid the disadvantages.

Prerequisites:

C-programming experience; basic microcontroller concepts advantageous.

Content:

Introduction: OO terminology; OO programming in C; introduction to C++ (EC++); C++ for embedded applications; analysis with UML; practical exercise (C and C++ programming in the OOP context, modeling an embedded application with UML, using an embedded target).

Duration: 5 days

MicroConsult Services

C++ Advanced Training

Training Objectives:

Using templates, exceptions and the Standard Template Library (STL); realization of advanced object-oriented concepts with C++.

Prerequisites: Knowledge as conveyed in our training "OOP with C++".

Content:

Exception handling; standard classes and functions; templates and Standard Template Library (STL); C++ background knowledge; concrete resource management; advanced object-oriented concepts.

Duration: 5 days

Advanced C-Training

Training Objectives:

Professional use of ANSI-C libraries and self-defined data types.

Prerequisites:

Profound C knowledge as conveyed in our training "The Programming Language C", or comparable C-programming experience.

Content:

Language elements (objects, object types, unions, structures, data types, type conversion); programming methods (object-oriented programming, recursive functions, complex types); C standard library (dynamic memory management, list/block operations, file processing, signal handling).

Duration: 5 days

MicroConsult Services

Coaching – Decisive Impulses at the Right Time

We're by your side at every phase of your project and provide competent and efficient support for your development teams. With a proven methodology and a high degree of practical orientation, we are in a position to identify development risks at an early stage. Make your projects transparent, predictable and cost-efficient. Minimize development times and costs and protect your developments. Focus on successful, high-quality project work.

Engineering – Workload Reduction at Your Fingertips

When projects exceed your own capacities, the MicroConsult experts take the weight off your shoulders. With our support, you focus your efforts and resources to achieve the most relevant results. Outsource what is not part of your company's core competence to our development specialists: from the definition and optimization of development processes, the development of hardware, software and system architectures to the test of the finished product.

The Authors



Thomas Batt

joined us in 1999. As trainer and project coach, he specializes, amongst others, in software development (tools, methods, processes) for embedded systems and real-time systems. In addition, our customers benefit from his many years of experience as a trainer for microcontroller platforms.



Frank Listing

joined us after 10 years of software development practice. Since 2002, he has been project coach focusing on Microsoft platforms, object oriented programming and embedded system test. He is responsible for .NET subjects and continuously shares his know-how in publications and technical articles. As counterbalance for his daily 'brain acrobatics', his hobbies are biking and inline skating – at maximum speed.



Peter Siwon

has seen many different aspects of the "embedded world": research, training, development, project management, sales and marketing. Today, he is general manager of MicroConsult. For more than 20 years, he has authored articles for technical journals. In addition, he was moderator and lecturer at countless events on the embedded scene. Water is his element. He is a passionate swimmer and canoeist, and water is also his favorite drink which is quite untypical of a Bavarian.

Imprint

Publisher:

MicroConsult GmbH
Charles-de-Gaulle-Str. 6
81737 Munich, Germany
Tel. +49 89 450617-0
Fax +49 89 450617-17
www.microconsult.com

Project Management and Editing:

Sabine Pagler
Marketing Coordinator
MicroConsult

Authors:

Peter Siwon
Thomas Batt
Frank Listing
MicroConsult

Translation:

Sabine Pagler

Trend Guide Media Partner:



www.elektronikpraxis.de



MicroConsult Microelectronics Consulting & Training GmbH
Charles-de-Gaulle-Strasse 6 • D-81737 Munich, Germany
Tel. +49 89 450617-71 • Fax +49 89 450617-17
E-Mail: info@microconsult.com • www.microconsult.com