Software Architectural Tactics and Patterns for Safety and Security

Christian Rehn

TU Kaiserslautern, 67663 Kaiserslautern, Germany,
c_rehn@cs.uni-kl.de

Abstract. Safety and Security are important quality attributes of today's software and their importance is even increasing. So it is necessary to address these aspects at the architectural level, although this is not sufficient to build safe and secure systems. Patterns and tactics enable reuse for this task. This paper presents the basic notions and explains why it's convenient to focus on tactics. Some examples show how safety and security are addressed. In the end the value and applicability of tactics and patterns for safety and security are discussed.

Table of Contents

1 Introduction

Nowadays Software is used not only in offices and living-rooms but also in safety-critical environments and for tasks where sensitive data or huge amounts of money are involved. On the other hand software becomes more and more networked, distributed and ubiquitous. So the risk of failing or compromised software increases as well as the severity of the possible consequences. Thus safety and security are major issues in software engineering and their importance is even increasing.

Unfortunately it is a complex task to build safe and secure systems. Especially there seems to be much software which should be secure, but isn't. Every now and then the media report about critical security holes in diverse software and insufficiently protected user data in online communities.

For safety the problem seems to be not that severe because fewer non-experts write software for safety-critical systems. But nevertheless building safe systems is evidently a complex task, too.

In software-engineering reuse is a major means of reducing development effort and increasing quality by using existing solutions that are known to be well engineered. At the software architecture level this is done by so-called patterns and tactics. This paper presents how these patterns and tactics address safety and security.

2 Basic Concepts

Before actually going into the topic, some notions have to be defined. Unfortunately some of them vary throughout the literature. By explaining the definitions and giving some examples the basic concepts are described.

2.1 Faults, Failures and Errors

There are quite complex systems of defect classification. Unfortunately they are not always compatible to each other. That means some notions are defined differently by different classification systems, c. f. [3], [1].

For this paper the following definitions are used:

Definition 1. *A **failure** is the "event that occurs when the delivered service deviates from the correct service."[3]*

Definition 2. *A **fault** is an "incorrect step, process, or data definition in a computer program"[1]. It can be internal (fault in code, specification, etc.) or external (wrong input data, attack, etc.)[3].*

Faults can cause failures but they don't have to. If the fault is contained by some tactic or the conditions for turning the fault into a failure are never met, no failure will be observable. Likewise external faults need internal faults in order to produce a failure. So they cannot be the sole cause of a failure.

Definition 3. *A **vulnerability** is an internal fault which makes an attack possible (c. f. [3]).*

Definition 4. *An **error** is a deviation in the systems' state from the correct state (c. f. [3]).*

Note that the occurrence of a failure doesn't necessarily mean that there is an error, since it is possible that the error is just there at the moment of the occurrence of the failure. A failure doesn't need to change the state of the system permanently. In contrast to that a fault is always a prerequisite for a failure.

A software flaw like an unchecked buffer is an internal fault. This can cause failures but won't if used only in the correct way (data of the correct length). In combination with an external fault (e. g. an attack) a failure can be produced, so the unchecked buffer is a vulnerability. When the failure has occurred, this means that an erroneous system state has been observed. This erroneous system state is the error, while the observation of the error (wrong output, wrong behavior, system downtime, etc.) is the failure.

2.2 Safety and Security

Definition 5. ***Safety*** *"is the ability of an item not to cause [. . .] unacceptable consequences during it's use."[6, p. 9]*

Definition 6. ***Security*** *"is the absence of unauthorized access to, or handling of, system state."[3]*

Security can be subdivided into the following parts [3]:

- **Confidentiality**: Security is low when an unauthorized person can access confidential data.
- **Integrity**: Security is low when an unauthorized person can alter confidential data or system state in general.
- **Availability**: Security is low when an attacker can decrease availability of the system (denial of service attack (DoS)).

There are other classification systems for security[1] different from the above shown "CIA triad" as well as other aspects of security, which can even vary with respect to the given domain. For example nonrepudiability can be a security aspect in an online shop system. There it is important that it's clear who made an order. On the other hand the opposite, plausible deniability, can be a security attribute too. For whistle blowers it might be very important that they can plausibly deny to have sent a particular message. Off the record messaging is an example for a technology that allows plausible deniability [9]. Those other—secondary—aspects are not addressed in this paper.

Fig. 1. Relationship between dependability and security (taken from [3]).

Another notion used in this context is *dependability*. This notion combines aspects of safety and security in the way depicted in figure 1.

[1] e. g. Microsoft's STRIDE model

2.3 Software Architecture

The notion of software architecture evolved in the early 90s [11], but the origins date back to the late 60s and early 70s, when the software crisis led to the discipline of software engineering. Most notably David Parnas pointed out the importance of system structure (c. f. [12]).

The essence of this, focusing on the structure or more precise the structures as there are many structures in a system, became the core of what is now called software architecture.

There are many definitions of software architecture which differ in detail. But all of them are about the mentioned system structures. The following definition taken from the book "Software Architecture in Practice" is widely used and thus will be used here:

Definition 7. *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."[4, p. 3]*

3 Patterns and Tactics

3.1 Patterns

Definition 8. *A **pattern** "describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate."[7]*

Patterns are normally not invented but discovered. Someone realizes that a recurring problem was solved the same way several times. This already is the pattern, the recurring solution to the recurring problem. It can then be named, formalized and documented in order to make reuse possible.

The notion of patterns originates from building's architecture. In 1977 the American architect Christopher Alexander wrote the book "A Pattern Language", which describes how to build towns and buildings. [2]

On Patterns he writes: "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." [2] This is the essence of the pattern notion and applies to almost every kind of pattern.

Ten years later Kent Beck and Ward Cunningham transferred Alexander's idea to software and used patterns to describe how to build graphical user interfaces in smalltalk [5].

After Kent Beck and Ward Cunningham had brought patterns to software, some research was done and patterns first became popular in 1995 when the book "Design Patterns: Elements of Reusable Object-Oriented Software" [10] was published. In this book Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the so called "Gang of Four", describe 23 patterns they found in object-oriented system designs.

Looking at the definition above the connection between software architecture and patterns is obvious. But there are not only patterns at the architecture or design level. Since 1987 many patterns have been discovered on different levels of abstraction. The Gang of Four concentrated on design patterns, Martin Fowler found analysis patterns and James O. Coplien discovered

so called idioms (language-specific implementation patterns). There are architectural patterns—often referred to as styles—and documented bad solutions so called anti-patterns. So a whole pattern community arose, there are people working on patterns[2] and conferences on patterns[3].

There are also patterns addressing architectural quality attributes like safety and security. An example for such a pattern is Triple Modular Redundancy (TMR) which addresses safety.

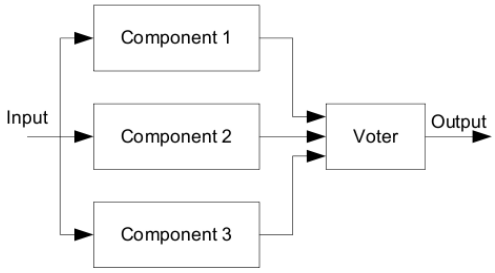

Fig. 2. A TMR pattern (taken from [16]).

The TMR pattern described in figure 2 involves three components and one voter and the input for all three components comes from the same source. This implies that there are two possible single points of failure: the data source and the voter.

In order to remove those single points of failure several variations of the TMR pattern could be used. Some employing several voters, some having several data sources and so forth. All of these TMR variations would share the basic idea of having three components and some kind of voting mechanism but would still still be different. In particular they would have different safety properties. [16]

3.2 Tactics

As described above there tend to be many variations of software architectural quality patterns. It would be more convenient to study the basic ideas of those patterns instead of the quite coarse-grained patterns itself, so the notion of tactics has been defined:

Definition 9. *A **tactic** is a design decision for realizing quality goals at the architectural level (c. f. [4, p. 100]).*

Tactics are rather simple ideas. They are fine grained but abstract and thus as opposed to patterns expressible in just a few sentences.

Although tactics are fine grained, they are not atomic. They can be refined, so there is a hierarchical structure of tactics [4, p. 100]. For example *redundancy* is a tactic which can be specialized by the tactics *replication*, *functional redundancy* and *analytic redundancy*.

Patterns tend to be much more complex, because they package several tactics and this in a more concrete way. So tactics are building blocks for patterns.

[2] c. f. Hillside Group (`http://hillside.net/`)
[3] PLoP, EuroPLoP, ChiliPLoP and others

The TMR pattern for example packages the redundancy and the voting tactics. Depending on the point of view you can say that refined redundancy tactics can be used for TMR or you can say TMR also packages the *diversity* tactic.

On the other hand a pattern doesn't need to package *several* tactics. Sometimes a pattern realizes just *one* tactic. For example the tactic *authenticate users* is described as follows: "Authentication is ensuring that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication."[4, p. 116] That is all. It's a simple idea explained in one sentence and a few examples.

The pattern *authenticator* realizes this tactic. [13] explains it in four and a half pages (pp. 323-327). But not only the description is more detailed. The pattern itself is much more concrete than the tactic. It defines roles by class diagrams, describes their behavior by sequence charts, mentions operating systems as a context, and discusses conflicting forces, variations and consequences.

The complexity of patterns as well as the fact that there are many different variations make it difficult to reason about them. An architect who has to decide which pattern to use has to take every variation into account.

So in order to understand software architectural quality patterns it is convenient to study the tactics they are composed of. Tactics also simplify reasoning as an architect just has to decide on a manageable amount of tactics and then composes a specific pattern out of them (c. f. [15]). Here the notion "pattern" is used in a broader sense as the patterns here are constructed rather than discovered which is not the original idea of patterns.

The simplicity of tactics make them easy to use but also lower their value to some extend. They tell what to do in order to reach safety and security but they don't tell how to do. Selecting and combining tactics, which can be quite complex tasks, are still the architect's task. This increases freedom to the cost of reusability.

For example if an architect chooses to use the tactics *authenticate users* and *authorize users*, these tactics tell what could be done. The architect could for example choose between authentication by user name and password, digital certificates or biometric identification. But this decision is not enough to build the architecture.

Patterns are much more useful here as they specify components or at least component roles. The architect can map these roles to the architecture. So patterns don't only make the reuse of ideas possible—as tactics do—but also the reuse of parts of the architecture, which is a much higher benefit.

As described above tactics are building blocks for patterns and although they might be less valuable for an architect they are worth studying because they carry the essence of the ideas behind safety and security patterns. Thus in the following two sections a collection of tactics for safety and security is introduced.

4 Safety Tactics

As stated in definition 5 safety is about reducing undesired consequences—accidents—to an acceptable level. An accident can happen when the software doesn't behave according to it's specification, which means there is a failure. Nevertheless the specification could be faulty itself but architecture is not the place to handle this. This would be subject to quality assurance in the specification process.

So in order to ensure safety, failures must be avoided, and, if a failure occurs, detected and contained, that means the consequences mitigated. In the

fallowing a—naturally non-exhaustive—catalog of safety tactics is presented and some of these tactics are explained in more detail.

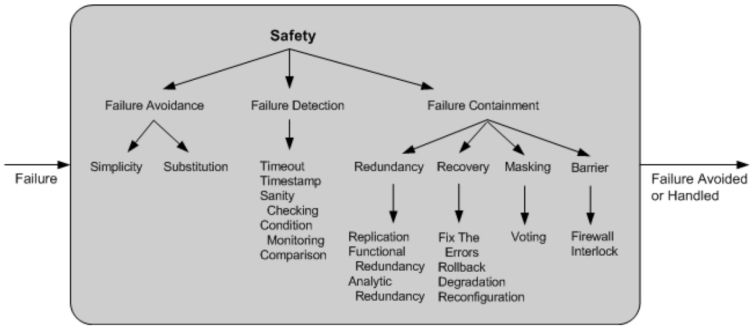

Fig. 3. Safety tactics (taken from [15]).

The tactics shown in figure 3 are the result of research done by Weihang Wu at the University of York. They can be found along with a detailed description in [15].

4.1 Failure Avoidance

One way to handle failures is to avoid them. This is not an easy task but should be the goal for engineering software and especially safety-critical software. *Simplicity* is a tactic for achieving this:

Simplicity

- Aim: Make the software simple in order to reduce faults.
- Description: One possibility to avoid failures is to make the system as simple as possible. Simple software with a small set of simple components with simple interfaces is less prone to faults and thus will have fewer failures so the software is more safe.
- Applicability: This is a rather general tactic used during the whole architecture design process but applying this tactic may be difficult.

4.2 Failure Detection

There is no 100% defect free software so it can be assumed that there will be failures in any non-trivial software system. In order to handle those failures it is inevitably necessary that those failures are detected. Tactics like *timeout* or *sanity checking* can do this.

4.3 Failure Containment

Failure Containment means mitigation of failure consequences. This can be done in several ways.

Redundancy enables *masking*, can lower the probability that a failure is not detected and generally enables the system to work in case of a component breakdown or malfunction. Another possibility can be the *recovery* from a failure. This can mean that the system can continue operation with reduced

functionality or quality (*degradation*) or that the problem can be completely resolved. By *masking* the failure can be prevented from propagating to the system boundary so that the failure only occurs at a component but the system can continue operating normally. Therefore a kind of redundancy is necessary.

Redundancy. Redundancy is a valuable but expensive tactic for realizing safety goals. There are three different kinds of redundancy: replication, functional redundancy, and analytic redundancy.

Replication

- Aim: Replicate components in order to detect hardware failures.
- Description: Replication is the simplest form of redundancy. It introduces one or more additional but identical components. [14]
- Applicability: Replication is the cheapest form of redundancy although in some cases (expensive hardware, constraints on size or weight, etc.) even this can be considered too expensive. On the other hand replicated components will have the same software flaws, so this form of redundancy is only beneficial if hardware failures are likely to occur.

Functional Redundancy

- Aim: Introduce independently implemented components in order to detect hard and software failures.
- Description: Functional redundancy allows for redundant components to be different in such a way that they might have different implementations and use different algorithms but will produce the same output if given the same input. This means that all implementations must be mathematically equivalent, so the diversity is only internal and not visible outside of the (sub)system. [14]
- Applicability: Functional redundancy is more expensive than replication as it requires different implementations. This applies to development as well as to maintenance. But in contrast to replication functionally redundant components differ in software which makes it possible to detect software failures. Functionally redundant components can, but don't need to, involve hardware dependency. In such a case constraints on size and weight must be taken into account.

Analytic Redundancy

- Aim: Introduce independently developed components in order to detect hard and software failures.
- Description: Analytically redundant components may produce different results for the same input as they only need to adhere to the same specification. They can be developed completely independent from each other, can use different—not necessarily mathematically equivalent—algorithms and data structures and run on different platforms. [14] For example one component can measure a value while another can compute it.
- Applicability: Analytic redundancy is very expensive in development and maintenance. Everything is done twice and in different ways. On the other hand this potentially allows to detect even more software failures. Analytically redundant components can, but don't need to, involve hardware dependency. In such a case constraints on size and weight must be taken into account.

Recovery. In some cases it is possible to recover from a failure if such has been detected. These tactics range from setting the system to a degraded sate with reduced functionality (*degradation*) to completely recover from the attack by going to a known to be safe state and retrying the action which led to the failure in hope it doesn't occur again (*rollback*).

Masking. By *masking* a failure can be prevented from propagating to the system boundary. One tactics for doing so is *voting*.

Voting

- Description: The voting tactic is used with some form of *redundancy* and introduces a voter component which decides which value to take. So possibly occurring failures are masked which means they are not observable at the system boundary any more so they are failures of the component but not failures of the whole system. There are several possibilities for such a voting algorithm. In particular there can be "trusted" components, which means in case of different values the value from the trusted component is taken and the others are discarded. There can also be majority decisions or the voter computes a kind of average value.
- Applicability: This tactic can only be applied if there is a form of redundancy. Majority decision voting can only be done if there is an odd number of redundant components/data, which means at least three. And for the trusted component algorithm there has to be such a trusted component and obviously a reason to trust this component more than the others. For the average value algorithm it must be sure, that a possibly erroneous value is not that different that it makes the computed average value erroneous too.

5 Security Tactics

Safety tactics, which were considered in section 4, are about preventing actions from unintentionally resulting in unacceptable consequences, whereas security tactics are about preventing action deliberately resulting in unacceptable consequences. This means there has to be an attacker, an unauthorized person, who wants to access or compromise the system or the data in the system.

There are different ways such a situation could be handled: Actually preventing the attack meaning preventing the attack from being successful is the most effective way but it's not always possible. If it's not, it might be possible to detect an ongoing attack. Normally such an attack needs some time and maybe enough time to detect and mitigate it. A third possibility is to recover from a successful attack and thus mitigate it's consequences.

In order to build a secure system the three aspects of security have to be considered: availability, confidentiality and integrity. Especially availability is different from the other two aspects and therefore different tactics need to be applied in order to maintain availability. There are also secondary security attributes (see section 2.2), which are not considered here.

The tactics listed in figures 4 and 5 were developed by the Software Engineering Institute at Carnegie Mellon University and are described in [4]. In the following an overview is given and some tactics are explained in more detail.

5.1 Resisting Attacks

Although not always possible resisting attacks is the most effective way to ensure security. Most notably authentication and authorization are tactics to

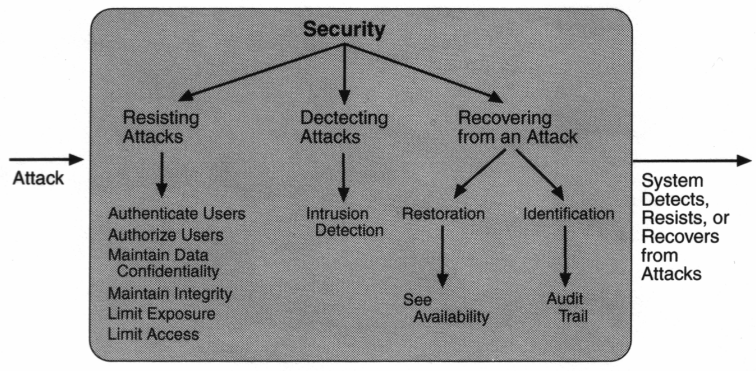

Fig. 4. Security tactics (taken from [4]).

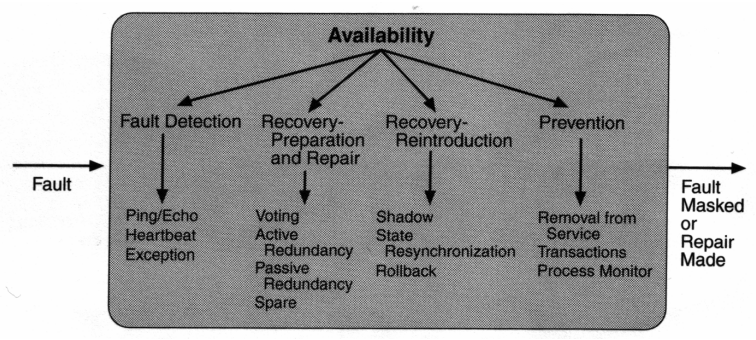

Fig. 5. Availability tactics (taken from [4]).

ensure confidentiality and integrity. When there are different users with different rights, these tactics are often inevitably necessary to be applied.

Authenticate Users

- Aim: Verify that a user is the person he or she claims to be.
- Description: One step to ensuring that access is only granted to authorized people is to ensure identity. So nobody can claim to be another person. Realizations range from knowledge (user name and password) and ownership (smart cards) to physical properties (biometric data: finger prints, etc.).
- Applicability: Some of these realizations require special hardware (finger print scanners, etc.). Another limitation is, that all possible identities must be known to the system. This is trivial for a central server but not for something like an embedded system in a car where every motor mechanic should have access to.

Authorize Users

- Aim: Specify which rights which users have.
- Description: Allow only persons (authenticated users) to access or modify certain data if they are allowed to. Authorization can be done for single users or groups of users.
- Applicability: Use this tactic always when there are users who should have different rights. Authorization is not necessary if everybody has the same rights. Note that giving everybody the maximum rights is not secure if it's not necessary because this would not ensure unauthorized access as unprivileged persons could use features not intended for them.

Limit Exposure

- Aim: Limit the possibilities for vulnerabilities.
- Description: Attacks often use single vulnerabilities in systems, so decreasing the number of services decreases the likelihood of a vulnerability on this host. There are just fewer possibilities for placing an attack. [8]
- Applicability: Some services are necessary, so removing them is not always an option. This tactic is especially useful at system boundaries as attacks often come from the environment and not from within the system. On the other hand applying this tactic on inner components too is also sensible, as an attacker already could have used a vulnerability at the system boundary to get into the system, and it is also possible for an attack to come from within the system as a less privileged person, who has (limited) access to the system, could try to gain access to confidential data.

5.2 Detecting Attacks

As there is no 100% security it is impossible to prevent all attacks in advance. But sometimes it can be helpful to detect an ongoing attack in order to be able to react or even to detect a successful attack in order to mitigate the consequences. Intrusion detection is a way to do that.

5.3 Recovering from an Attack

When an attack has been successful it is necessary to react properly. The attack could be detected automatically by an intrusion detection system or manually by users recognizing that the system is not available or data is corrupted or

modified in a suspicious way. Successful confidentiality violations are often not detected but sometimes the attacker may try to blackmail someone with the "stolen" data. This would also result in the need for a proper reaction. Tactics for recovering from an attack describe what could be done.

Restoration. In case of a successful attack the reaction depends on what the attacker has done. When there is a confidentiality problem, restoration is not necessary as the system itself remains unchanged. For integrity a kind of backup might be helpful. For availability several tactics are mentioned in [4]. There are four main aspects: failure detection, recovery preparation and repair, recovery-reintroduction, and prevention (see figure 5).

Failure detection ([4] calls it "fault detection" but according to definitions 1 and 2 it's failure detection) is in this case about detecting if a component is still operating. One example would be the *ping/echo* tactic which detects unavailable components by repeatedly sending ping signals awaiting an answer.

Recovery preparation and repair focuses mostly on kinds of redundancy in order to keep the system available when an attacker manages to disable one component (or even more). This is closely related to recovery-reintroduction. These tactics mainly specify how to handle redundant components.

Finally there are also some tactics which try to prevent availability problems but as this is hardly possible in general, these tactics can only focus on certain aspects.

Some of those tactics are also safety tactics for example *voting* or *rollback*. Here the only difference is the point of view which shows that tactics are not tied to a single quality goal.

Identification. Some attacks might be successful despite all prevention tactics applied. In such a case it can be useful to identify the attacker. An *audit trail* which is a kind of log file can help doing so.

6 Applicability

When looking at the tactics presented in this paper the question arises if they are applicable in every case. Of course it's not like that. Most of them seem to fit for large distributed systems. The *ping/echo* tactic for example can be used when a system is distributed over several network nodes but is apparently useless for building desktop applications.

Redundancy is also suitable for large distributed systems. Embedded systems are another domain where it can be used. For example some values could be computed or measured. Having components for both approaches would be a form of *analytic redundancy*. But again for desktop applications *redundancy* of any kind is in most cases either useless (e. g. *replication* of software components) or much too costly.

There is only a relatively small amount of tactics suitable for medium-size desktop applications. This includes *simplicity*, forms of *degradation*, *authenticate users*, *authorize users* and *audit trail*. For small web-applications like weblogs the list of appropriate tactics would not be much longer.

Naturally any list of tactics will be incomplete as there can always arise new ideas but the lack of tactics for such applications is a systematic problem and not a problem of the presented list. Safety and security can be best addressed at the architectural level when the system is large, complex and distributed.

But large complex distributed systems are not the only ones that need to be safe and secure. Every system used in a safety or security-critical environment has to be safe respectively secure. This applies even to very small programs such as Unix commands.

The smaller a system the more other factors are important for safety and security. These are not solely architectural issues. Architecture is just one aspect for making systems safe and secure. Requirement analysis, implementation, verification and validation are other aspects.

Many vulnerabilities for example are created during the implementation. Unchecked user input, possibilities for buffer overflows and insecure encryption (e. g. missing salt for password hashing) are still major security risks and cannot be satisfactorily addressed at the architectural level.

Similar for safety. A good architecture alone doesn't prevent components and systems from failing. Hence it is necessary to have a whole safety respectively security concept and not just a few tactics. Such a concept can comprise architectural means as well as an adapted process and even personnel management and education. And of course this concept is depending on the system that is intended to be built.

7 Conclusion

Safety and security are important challenges in software engineering. As software becomes more and more ubiquitous, distributed and networked, it needs to be increasingly dependable, which means in particular safe and secure.

Software architecture is one level at which safety and security have to be ensured. As described in section 3.2 tactics help to simplify the reuse of architectural knowledge by specifying building blocks for architectural quality patterns.

There are a number of tactics addressing safety and security and these tactics are well known and well structured. So there is the possibility for reuse of ideas as tactics mainly structure ideas for design decisions.

There are also a number of patterns for safety and security. These are much more complex and tend to have many variations whereas each variation may have different safety and security properties. This makes it difficult to reason about them. On the other hand they allow much more reuse and are thus more valuable than tactics.

But architecture is not the only level at which safety and security need to be addressed. Other factors influence safety and security too. Many vulnerabilities are for example implementational issues. So there has to be a whole safety respectively security concept which depends on the system to be build.

So it's not enough to keep an eye on safety and security while building the architecture but it can be vital for building safe and secure systems. And in the end it is also important to know that that there is neither 100% safety nor 100% security.

References

1. IEEE Standard Glossary of Software Engineering Terminology (1990), `http://ieeexplore.ieee.org/ISOL/standardstoc.jsp?punumber=2238`
2. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language. Oxford University Press (1977)
3. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1, 11–33 (Jan 2004)

4. Bass, L., Clemens, P., Kazman, R.: Software Architecture in Practice. SEI Series in Software Engineering, Addison-Wesley, 2 edn. (2003)
5. Beck, K., Cunningham, W.: Using Pattern Languages for Object-Oriented Programs. In: OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming (1987), http://c2.com/doc/oopsla87.html
6. Birolini, A.: Reliability Engineering: Theory and Practice. Springer, 5 edn. (2007)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, vol. 1: A System of Patterns. John Wiley & Sons (1996)
8. Ellison, R.J., Moore, A.P., Bass, L., Klein, M., Bachmann, F.: Security and survivability reasoning frameworks and architectural design tactics (2004), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.61.3340
9. Engel, T.: Sichere Nachrichtenübermittlung mit Off-the-Record-Messaging. Diplomarbeit, Technische Fachhochschule Berlin (2007), http://public.tfh-berlin.de/~s30935/off-the-record-messaging.pdf
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
11. Garlan, D., Shaw, M., Okasaki, C., Scott, C.M., Swonger, R.F.: Experience with a Course on Architectures for Software Systems. In: in Proceedings of the Sixth SEI Conference on Software Engineering Education, Springer-Verlag, LNCS 376. Springer Verlag (1992), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.638
12. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)
13. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons (2006)
14. Sha, L., Gagliardi, M., Rajkumar, R.: Analytic Redundancy: A Foundation for Evolvable Dependable Systems. In: Proceedings of the 2nd ISSAT International Conference on Reliability and Quality of Design. Software Engineering Institute Carnegie Mellon University (1995), http://www.sei.cmu.edu/publications/documents/95.reports/95.tr.005.html
15. Wu, W.: Safety Tactics for Software Architecture Design. Master's thesis, The University of York (2003)
16. Wu, W., Kelly, T.: Safety Tactics for Software Architecture Design. In: Proc. 28th Annual International Computer Software and Applications Conference COMPSAC 2004. pp. 368–375 (2004)