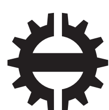
Tampereen teknillinen yliopisto. Tietotekniikan laitos. Raportti 2
Tampere University of Technology. Department of Pervasive Computing. Report 2

Veli-Pekka Eloranta, Johannes Koskinen & Marko Leppänen (eds.)

# Proceedings of VikingPLoP 2013 Conference



TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Tampereen teknillinen yliopisto. Tietotekniikan laitos. Raportti 2
Tampere University of Technology. Department of Pervasive Computing. Report 2

Veli-Pekka Eloranta, Johannes Koskinen & Marko Leppänen (eds.)

# Proceedings of VikingPLoP 2013 Conference

Tampere University of Technology. Department of Pervasive Computing
Tampere 2013

ISBN 978-952-15-3166-8 (printed)
ISBN 978-952-15-3167-5 (PDF)
ISSN 2323-9174

# Preface

This is the proceedings of VikingPLoP 2013 – a record of all papers workshopped during the conference. VikingPLoP is a Nordic conference of pattern languages of programs which took place this year in Horse Inn of Luomajärvi, Ikaalinen, Finland in March 2013. VikingPLoP was organized jointly by Tampere University of Technology and Hillside Europe. VikingPLoP 2013 was also sponsored by Wiley which provided books for the focus group reading session. The conference was organized in Finland for the second time in a row. Previous location in 2012 was in Saariselkä Lapland. In 2013 vikings were moving towards south and chose the Horse Inn in Ikaalinen as the venue as it offered a luxurious opportunity for participants to experience rustic romance, good food, horseback riding, traditional Finnish sauna, the nature, and wilderness tracks. In March the landscape was still covered in snow making the landscape ruggedly beautiful.

The papers in this proceedings book are updated versions of the papers workshopped in the conference. In the beginning, participants submitted their papers for shepherding process. In the shepherding process, the shepherd, an experienced pattern writer, gave ideas and feedback for the author, colloquially known as a sheep. The sheep incorporated this feedback in to her paper. After three iterations of shepherding the paper was discussed at the conference in a writer's workshop. The workshop group gave comments, criticism and praise. After the conference the authors updated their papers according to the workshop feedback.

This process of giving feedback was made possible by having a community of trust. Mutual trust was built by playing non-competitive games and by having social activities. VikingPLoP 2013 focused on patterns and their usage in various fields of expertise. These fields included a wide range of topics from educational patterns to safety patterns and embedded system's software architecture patterns. Bringing people together from various fields of expertise stimulates creativity and new ideas might emerge. These innovations are reflected in the papers in these proceedings. VikingPLoP 2013 was especially a conference for newcomers and over half of the participants were first time PLoP participants.

These proceedings contain 9 papers. In addition, a book reading workshop was arranged with Bob Hanmer who presented his new title Pattern-Oriented Software Architecture for Dummies and discussed it with the participants using video conferencing tools.

As expected, VikingPLoP 2013 was an enjoyable and fun experience. We are grateful for all contributors for your involvement. If you wish to participate in VikingPLoP in the future, please come and find out more information about the next conference http://www.vikingplop.org and join the community.

We wish that these proceedings are a valuable source of information in your efforts. We hope that you will enjoy reading the following pages.

October 2013
The program chairs,
Veli-Pekka Eloranta and Marko Leppänen

# Thanks

We would like to send out our thanks to everybody who has helped us to organize this event:

- the authors, who have written and submitted their papers to the conference;
- the shepherds, whose feedback to the submitted papers have helped the authors to be accepted to the conference;
- the program committee, whose help has been invaluable;
- John Wiley & Sons, Inc., which sponsored us by providing book to discuss in the focus group
- Staff of Horse Inn of Luomajärvi for making the experience smooth and memorable!

**Organization of VikingPLoP 2013**

VikingPLoP is a non-profit event organized by Tampere University of Technology with support from Hillside Europe and various individuals from pattern community.

**Program chairs**

- Veli-Pekka Eloranta, Tampere University of Technology, Finland
- Marko Leppänen, Tampere University of Technology, Finland

**Program committee**

- Ville Reijonen, Kauppalehti Online Development, Finland
- Johannes Koskinen, Tampere University of Technology, Finland
- Juha Pärssinen, VTT, Technical Research Centre of Finland, Finland
- Kai Koskimies, Tampere University of Technology, Finland
- Dirk Schnelle-Walka, Technische Universität Darmstadt, Germany

**VikingPLoP supporters**

- Tampere University of Technology - Department of Pervasive Computing
- Hillside Europe
- John Wiley & Sons, Inc.

**Shepherds**

- Marko Leppänen
- Johannes Koskinen
- Christian Köppe
- Bob Hanmer
- Dirk Schnelle-Walka
- Jari Rauhamäki
- Veli-Pekka Eloranta
- Ville Reijonen

# Table of Contents


Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems ........................................ 1

*Pekka Alho and Jari Rauhamäki*

Patterns for Operating the Control System Remotely ........................................ 18

*Veli-Pekka Eloranta*

Patterns for Decoupling Hardware and Software ........................................ 30

*Johannes Koskinen*

Patterns for Designing Programming Assignments ........................................ 48

*Samuel Lahtinen*

Patterns for Messaging in Distributed Machine Control Systems ........................................ 63

*Marko Leppänen*

Catalog of Safety Tactics in the light of the IEC 61508 Safety Lifecycle ........................................ 79

*Christopher Preschern, Nermin Kajtazovic and Christian Kreiner*

Patterns for safety and control system cooperation ........................................ 96

*Jari Rauhamäki, Timo Vepsäläinen and Seppo Kuikka*

A pattern for bootstrapping ........................................ 109

*Ville Reijonen*

Probabilistic Dialog Management ........................................ 114

*Dirk Schnelle-Walka, Stefan Radomski, Stephan Radeck-Arneth*

# Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems

Pekka Alho[1], Jari Rauhamäki[2]

[1] Tampere University of Technology, Dept. of Intelligent Hydraulics and Automation, Finland
pekka.alho@tut.fi
[2] Tampere University of Technology, Dept. of Automation Science and Engineering, Finland
jari.rauhamaki@tut.fi

## 1 Introduction

Distributed control systems are continuously gaining importance, as more and more devices and machines are equipped with embedded systems that control their operation. These controllers are increasingly more powerful and networked, providing intelligence and interoperability for the control system. Examples of such systems range from large mobile machines to robots and intelligent sensor networks. These systems often interact with physical processes, influencing many parts of our lives either directly or indirectly. Therefore they need to be *dependable,* which can be measured with the attributes of availability, reliability, safety, integrity and maintainability [1]. However, with the increased functionality and intelligence, the complexity of these systems is also increased, meaning that the development process becomes more demanding and dependability becomes more costly to achieve and verify. Another significant requirement of these systems is that they usually are real-time systems, which may put limitations on the architecture.

Many critical systems that have failed catastrophically are well-known – examples such as Therac-25 radiation therapy machine and the explosion of Ariane 5 rocket are infamous, whereas highly reliable systems receive little recognition, even though their study might give valuable ideas for the design and architecture of new software. One example of such systems can be found in telephony applications, namely Ericsson AXD301 ATM switches that achieved nine nines (99.9999999%) service availability, running software written in Erlang [2]. Erlang's highly decoupled actor model and fault handling based on supervisors have inspired especially LET IT CRASH and SERVICE MANAGER patterns found in this paper.

This paper presents three software patterns that can be used to improve control system dependability by implementing a decoupled architectural design with supporting fault handling. The decoupled architecture can also be used to introduce additional fault tolerance solutions – like checkpointing and rejuvenation – gradually to the system, until a sufficient level of reliability has been achieved [3]. Our patterns have been encountered originally in research of remote handling control systems used to teleoperate robotic manipulators, but all patterns have examples of other known uses as well. Some of these examples are presented in the corresponding sections of the patterns.

## 2 Context of the Patterns

Fault tolerance cannot be implemented without redundancy of some kind. To have fault tolerance for e.g. computer failures, we would need at least two computers – if one fails the other one can detect the error and try to correct it. Software faults on the other hand are typically development faults, which are harder to detect and correct than hardware faults. To have good coverage for software faults, we would need diverse redundancy, but even this form of fault tolerance has been criticized of being susceptible to common mode failures [4]. Development costs for design diversity are also often seen as prohibitive.

The patterns in this paper present an alternative approach to fault tolerance, based on dividing the system into highly decoupled modules and using this to implement lightweight form of fault tolerance. We present an architectural pattern for this called DATA-CENTRIC ARCHITECTURE but this is of course not the only way to achieve a high level of decoupling. One of the key points of decoupling is that it should by itself improve reliability by limiting fault propagation and improving modularity and understandability of the system. In a way, modular approach can be seen like compartmentalization of ships – without compartments, every leak can sink the ship. An example of a software system that uses modularity to successfully implement fault isolation and resilience is the MINIX operating system, based on the idea of microkernel [5].

Modular and decoupled architecture can also be used to implement other reliability-improving patterns like SERVICE MANAGER and LET IT CRASH documented in this paper or other well-known patterns like LEAKY BUCKET COUNTER [6], WATCHDOG [6] [7], etc. The patlets of the patterns presented in this paper are listed in the **Table 1**. List of all referenced patterns with short descriptions can be found in an appendix.

**Table 1.** Patlets

| Pattern | Description |
|---|---|
| DATA-CENTRIC ARCHITECTURE | How to implement reliable and scalable distributed control system? Use data-centric middleware based on PUBLISH/SUBSCRIBE model [8] to reduce level of coupling between modules. |
| SERVICE MANAGER | How to detect faults and restart modules or processes after a failure? Implement a service manager that can monitor, start and stop modules. |
| LET IT CRASH | How to react to failures without crashing the whole system? Flush the corrupted state by "crashing" the process instead of writing extensive error handling code. Let some other process like service manager do the error recovery e.g. by restarting the crashed process. |

The presented patterns work together by building on the top of features provided by the higher abstraction level patterns as shown in **Fig. 1**, but all of the patterns are also typically used separately and in contexts other than distributed control systems.

The DATA-CENTRIC ARCHITECTURE provides the decoupled architectural model needed to use LET IT CRASH for fault handling. The SERVICE MANAGER pattern provides a way for trying recovery after failures, in addition to providing error detection and monitoring.

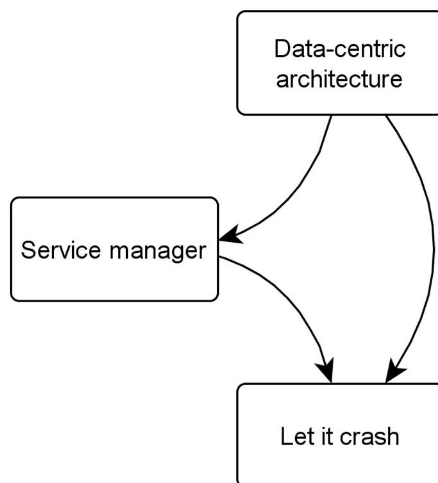
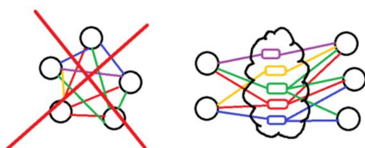



**Fig. 1.** Graph of pattern relationships

The idea of crashing a process suggested by LET IT CRASH may sound like a risky action to take. However, the idea is to offer recovery from transient physical and interaction faults (sometimes called Heisenbugs), ability to keep the system as a whole functioning, even if some internal process would crash, and possibility to hot-swap code and bug-fixes. The downside of this approach is of course that it is not suited for fail-operate systems like flight controllers that must be operational all the time – this type of systems would be the right domain to apply design diversity.

# 3 Patterns

## 3.1 Data-Centric Architecture



**Intent.** Implement an architecture based on decoupled modules (e.g. services, components or processes) that are connected with data-centric middleware.

**Context.** You are developing a distributed control system that consists of several subsystems and needs to interact with other heterogeneous systems like mobile machines or plant systems. The system has CPU and memory resources available to run an operating system – rather than being based on a basic time-triggered scheduler used in resource-constrained embedded systems. Failures in control functions (e.g. boom or manipulator control) may cause damage to the environment and equipment, meaning that some subsystems may be categorized as safety-critical.

**Problem. How to implement a reliable and scalable distributed control system?**

**Forces.**

- *Throughput*: Some time-critical data like sensor measurements may be updated with short period, producing large amounts of communication.
- *Scalability*: New nodes and subsystems can join the system any time; assumptions about interfaces between modules should be minimized.
- *Changeability*: System configuration and functionality might change. Changing interfaces in a tightly coupled system requires code changes at both ends (and at all clients), so assumptions about expected behavior should be minimized. Point-to-point protocol based client-server architectures (like sockets or remote method invocation) are not ideal because of complexity and coupling introduced.
- *Maintainability and long expected life-cycle*: The control system has long expected lifetime and needs to be maintainable and extensible in the future – if subsystems are added or substituted, changes to existing modules need to be minimized. System should be easy to understand and modify without breaking it.
- *Reusability*: Same modules could be used in other control system implementations.
- *Interoperability*: Distributed control systems consist of and/or need to communicate with heterogeneous platforms.
- *Testability*: Tightly coupled modules are difficult to test because they are more dependent on other modules.
- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures.
- *Reliability*: A single fault in the control system software should not endanger functionality of the whole system (i.e. no single point of failures).
- *Real-time performance*: Control system interacts with the real world and needs to react in a deterministic manner.
- *Safety*: Need to detect if a module has crashed or is down (not releasing new information) so that the system can enter SAFE STATE [7] in a controlled fashion. Safety-critical and non-safety-critical subsystems cannot be tightly coupled, since errors may propagate.

- *Quality of service*: Different subsystems may have different requirements for quality of service[1] (QoS) policies. There is an impedance mismatch between e.g. real-time control systems that operate on a timescale of milliseconds and enterprise/high level systems that are several orders of magnitude slower.

**Solution. Use data-centric middleware based on PUBLISH/SUBSCRIBE model to reduce level of coupling between modules.**

Implement data exchange between modules by adopting a middleware that publishes data to a global data space instead of sending point-to-point messages or remote procedure calls; data-centric architecture is based on removing direct inter-module references by exposing the data and hiding the code. Management of the global data space is externalized to the middleware that implements a topic-based PUBLISH/SUBSCRIBE model. The middleware acts as a single source of up-to-date state information in the system, instead of applications managing state separately.

Modules do not need to know recipients of the data when publishing it, which reduces coupling. Instead of sending data directly to a recipient, it is published to a topic. Data can be e.g. sensor measurements, events or commands, but it must follow a shared data model which is represented as topics in the actual system implementation. Publishers register as data writers to a topic and interested subscribers can join the topic as data readers. The middleware automatically discovers new readers and writers, which means that new nodes can join the system on the fly. Single topic can have multiple readers and writers, as shown in **Fig. 2**. Moreover, a topic can have multiple instances, which are identified by a key value.

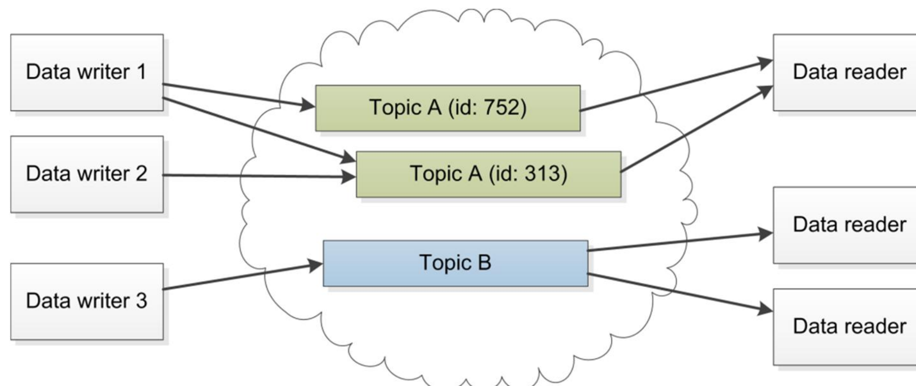



**Fig. 2.** Data is published to topics that can have multiple data writers and readers. Topic A has two instances, identified by the id number key value.

Expose data and hide the behavior. Instead of designing interfaces for components, you must design how to represent the state of the system and the external or internal

[1] QoS policies provide the ability to specify various parameters like rate of publication, rate of subscription, reliability, data lifespan, transport priority, etc. to control end-to-end connection properties. Policies can be matched on a request vs. offered basis.

events that can affect it. This needs a common data model, which captures the essential elements of the physical system and application logic. Conceptually the data model is similar to class diagram in object-oriented programming, since it consists of identifying entity types, which have data attributes assigned to them, and associations. The difference is that the data model focuses exclusively on data and not behavior.

Separate communication and application logic. Delegate network communications to a "data bus" formed by the publish/subscribe middleware (**Fig. 3**), so that the application logic can focus on the core functionality. Middleware takes care of maintaining the data up-to-date, automatically updating new nodes that join. If the middleware uses a central server as a message BROKER [8], it becomes a single-point-of-failure and possibly a bottleneck. Therefore, choose a decentralized middleware solution if possible to avoid this problem

Define appropriate QoS attributes for the data topics (reliability, durability, deadlines, etc.). Middleware manages the data lifecycle according to the associated QoS policy and matches policies offered by publishers vs. policies requested by subscribers.

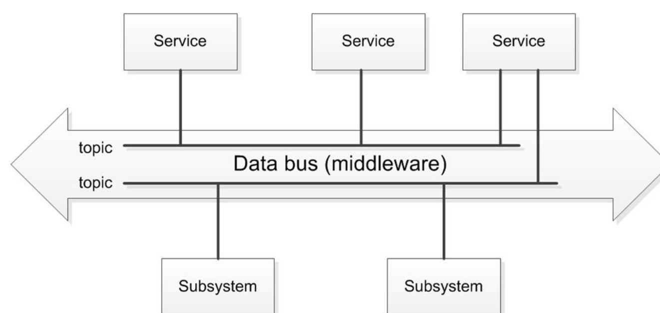



**Fig. 3.** Middleware implementation as a data bus that has no central components or brokers. Services and subsystems can join topics as publishers and/or subscribers.

Choose module granularity (size of the communicating modules) so that performance is not compromised. On the other hand, too large modules size may diminish the benefits of the data-centric architecture. The communication participants can be e.g. subsystems, applications, processes or modules, depending on the environment. Note that communicating modules can also exist on a same computer and use the data bus to get benefits of loose coupling locally.

Compared to message-centric publish/subscribe, the difference in data-centric model is that middleware "messages" – i.e. topic samples – are transparent to the middleware. In message-centric model, middleware does not know or care about message contents. With smart, QoS-aware data-centric middleware, application components can be leaner and take less time to develop because the logic that implements the QoS functionality is pushed down into the middleware. The application specifies these policies to the middleware during launch and it gets notified by the middleware during operation when QoS requirements are not being met.

**Consequences.**
+ Publishers do not need to know about subscribers.
+ Middleware provides interoperability between heterogeneous platforms
+ Decoupled design provides error confinement and other benefits like improved maintainability.
+ Modules can be changed dynamically because late joiners receive new data automatically; ability to hot-swap code can be easily implemented.
+ Network transport layer is abstracted as communications are externalized to middleware, which reduces communication related code and simplifies implementation.
+ Gives developers control of data delivery with QoS management; QoS can be used e.g. to guarantee reliable delivery eventually or that available data is kept up-to-date with best effort. Former would be useful for sending status changes or commands whereas latter could be used for sensor measurement for which guaranteeing delivery of old information makes no sense.
+ Reusability is improved since modules are using shared memory and have their own namespaces, etc.
+ Publish/subscribe based middleware scales effectively since recipients for data are not explicitly defined.
+ Performance gains can be achieved on multi-core machines since modules can be easily parallelized and they communicate asynchronously.
+/- Needs good and consistent data-models that must be managed and maintained.
- Sending of commands is not as straightforward as in client-server architectures since commands need to be parsed from the data. Serialization and deserialization of the data structures for transmission may also add overhead.
- Parsing of data complicates debugging because it adds another potential source for faults. If data is parsed incorrectly, it may not be self-evident where the fault originates.
- Errors in the middleware itself might complicate testing and be hard to detect.
- Middleware solution adds some overhead to message size and uses system resources.
- Possible vendor lock-in to the middleware provider.

**Examples.** Data Distribution Service for Real-Time Systems (DDS) is decentralized and data-centric middleware based on the publish/subscribe model. DDS is aimed at mission-critical and embedded systems that have strict performance and reliability requirements. Therefore, its implementations have typically been optimized and tested to suit the needs of these systems. DDS is used as the information backbone in the Thales TACTICOS naval combat management system that integrates various subsystems like weapons, sensors, counter measures, communication, navigation, etc. to a "system of systems". Applications are distributed dynamically over a pool of computers in order to provide combat survivability and avoid single-point-of-failures. System configuration can be adapted for use in various mission configurations, on-board & simulator training, and different ship types.

**Related Patterns.** BUS ABSTRACTION [7], and PUBLISHER-SUBSCRIBER.

MEDIATOR [9] increases decoupling in a similar fashion, but is designed to decrease connections between objects locally.

### 3.2 Service Manager

**Also Known as.** SUPERVISOR or SERVICE GATEWAY.

**Intent.** Service manager starts, stops, and monitors processes locally and takes care of resource allocation for systems that need high availability and real-time performance.

**Context.** You are developing a system with highly decoupled architecture (e.g. using DATA-CENTRIC ARCHITECTURE) that consists of large number of processes or tasks (services). These processes have dependencies and therefore need to be started in specific order. Process composition may change dynamically during runtime because your system will have intelligent functionality, it needs to adapt to new situations, or different functionalities need to be tested without stopping/restarting the whole system.

You know rough upper-limit estimates for how much system resources like memory and CPU time the processes will use.

The system has long expected life-cycle. It is likely to be deployed on a remote location like a forest or a control cubicle, making direct physical interaction with the system a bothersome task.

If you have a real-time operating system and a task gets stuck in a while loop or some other control structure, it freezes the whole system as other lower priority processes (including input devices and network connections) cannot get CPU time. In this case, the only option is usually to restart the whole computer manually.

**Problem. How to ensure that all dynamic modules in your control system are running correctly and you have enough system resources to achieve deterministic real-time performance?**

**Forces.**

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, in order to able to use other parts of the sys-

tem that are not connected to the failed subsystem. The system must detect faults and try to recover from them automatically. If a failure needs immediate reaction from a human operator, the system will not scale cost-efficiently and reliably.
- *Data logging/testability*: If a process fails, the failure should be detected and logged.
- *Real-time performance*: The control system needs to respond in a deterministic and predicable manner. Predictability includes system behavior when a fault is triggered.
- *System resources*: Control systems are typically deployed on embedded devices that have limited memory and CPU resources available. They may need to be monitored in order to guarantee the real-time performance of the system.

**Solution. Implement a service manager that can monitor, start and stop modules.** Create a local parent process (the service manager) that is responsible for starting, stopping and monitoring its child processes. The basic idea of the service manager is to keep its child processes alive by restarting them when necessary. Location of the service manager is on the same computer as the child processes in order to keep implementation simple. Therefore, all computers in the system need their own, independently functioning, service managers. The service manager is given the highest process priority in the system or put in the kernel so that a faulty real-time process cannot prevent it from functioning by consuming all available CPU time.

Start the child processes based on a fixed order or a dependency table read from a configuration file, similar to START-UP MONITOR [7], and/or implement a user interface that can be used to start and stop processes.

Use the service manager to allocate resources like CPU time and memory for the child processes and monitor their use. Expected maximum resource consumption can be specified in the same configuration file that is used for starting services. New processes are not started if there are not enough resources available. If a process consumes more resources than expected, it can be restarted, leading to error handling according to the LET IT CRASH pattern. Resource use can be followed e.g. with proc filesystem or *getrusage* call in Unix-like systems.

Since the key functionality of service manager is to monitor processes for failures, error detection can be based on additional or alternative techniques besides resource monitoring. This can be done with e.g. operating system features, HEARTBEAT [6] [7] or WATCHDOG.

If the service manager is deployed on a system that uses DATA-CENTRIC ARCHITECTURE, service startup interfaces can be implemented through the middleware. Since the middleware abstracts the location of the data, it can be used to remotely start dependencies. Example: service manager SM_A must start a service called S1. However, it has a dependency called S2 which cannot be found locally, so the service manager publishes a start request for S2. A second service manager SM_B on another computer notices the request, starts S2 and publishes information about the successful startup. SM_A receives information that S2 is available and starts S1.

The implementation for service manager needs to be kept fairly simple, since it acts as a single point of failure locally. This conflicts with the need to use of configu-

ration files, making resource checks, and providing user interface, so they should be based on external components or libraries that have been proven in use.

**Consequences.**

+ Detects and initializes recovery from transient faults that cause a process to consume too much system resources or become unresponsive. If the fault is persistent, LEAKY BUCKET COUNTER can be used to limit the number of restarts.

+ Ensures other processes stay alive and have sufficient resources.

+ Simplifies starting procedure of complex system that consists of large number of processes, making possible to start and stop a large number of processes automatically and in a specific order.

+ Cost-efficiency: the same service manager implementation can be reused on several systems.

+ Supports logging and reporting of errors so that they do not go undetected.

- Cannot detect faults that cause erroneous output for monitored components.

- Cannot recover persistent faults like development and physical faults, e.g. computer failures.

- Potential single point of failure that may stop the entire system from working if services are incorrectly terminated.

- Restarting a service may cause the system to behave in non-deterministic way and miss deadlines, which is a failure for a hard real-time system. However, it should be noted that the failure would have likely cause the system to miss the deadlines or exhibit some other unwanted behavior in the first place.

- Resource utilization needs to be estimated for the processes in order to set limits.

- Service manager uses system resources and may reduce performance.

**Examples.** The MINIX, a POSIX conformant operating system, based on a microkernel that has minimal amount of software executing in the kernel mode. The rest of the operating system runs as a number of independent processes in user mode, including processes for the file system, process manager, and each device driver. The system uses a special component known as the driver manager to monitor and control all services and drivers in the system [5]. Driver manager is the parent process for all components, so it can detect their crashes (based on POSIX signals). Additionally the driver manager can check the status of selected drivers periodically using HEARTBEAT messages. When a failure is detected, the driver manager automatically replaces the malfunctioning component with a fresh copy without needing to reboot the computer. The driver manager can also be explicitly instructed to replace a malfunctioning component with a new one.

Open source tool Monit (http://mmonit.com/monit/) can function as a service manager in non-real time systems. Following code listing shows an example configuration for Spamassassin daemon that restarts the daemon if its memory or CPU usage exceeds 50% for 5 monitoring cycles:

```
check process spamd with pidfile /var/run/spamd.pid
   start program = "/etc/init.d/spamd start"
   stop  program = "/etc/init.d/spamd stop"
   if 5 restarts within 5 cycles then timeout
   if cpu usage > 50% for 5 cycles then restart
   if mem usage > 50% for 5 cycles then restart
   depends on spamd_bin
   depends on spamd_rc
```

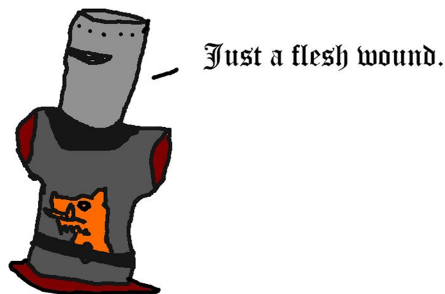
**Related Patterns.** FAULT OBSERVER [6], HEARTBEAT, SAFE STATE, SOMEONE IN CHARGE [6], START-UP MONITOR, STATIC RESOURCE ALLOCATION [7], and WATCHDOG.

To see how to design an application in a way that it can be easily restarted at any time, see LET IT CRASH.

MANAGER design pattern [10] can be used to manage multiple objects of same type – the idea is similar to SERVICE MANAGER (keep track of entities and provide unified interface for them) but the MANAGER focuses on different scope, i.e. managing entities (objects) of the same type and does not include resource monitoring or fault detection.

SYSTEM MONITOR [6] can be used to study behavior of system or specific tasks and make sure they operate correctly, e.g. by using HEARTBEAT or WATCHDOG. If a monitored task stops, SYSTEM MONITOR reports the error. Compared to it, SERVICE MANAGER has a more active role in managing the tasks.

### 3.3 Let It Crash




**Also Known as.** CRASH-ONLY [11], FAIL-FAST, LET IT FAIL or OFFENSIVE PROGRAMMING.

**Intent.** Avoid complex error handling for unspecified errors. Instead, crash the process and leave error handling for other processes in order to build a robust system that handles errors internally and does not go down as a whole.

**Context.** You are developing a distributed control system that consists of several processes and subsystems that need to cooperate to complete tasks.

DATA-CENTRIC ARCHITECTURE or some other asynchronous decoupled architectural design has been utilized so that processes are not using shared memory.

Some subsystems might have safety-critical functionality, but it is possible to move the system to SAFE STATE (i.e. the system is fail-safe type, not fail-operate). The system has dynamic state information from the user inputs and working environment in the process memory, e.g. tool tracking data in the case of a robot manipulator. This state data needs to be recovered after a failure.

The system has a mechanism like monitoring layer, supervisors or a restart manager for restarting the processes. This can be implemented at operating system, programming language or framework level, e.g. with the SERVICE MANAGER.

**Problem. How to implement lightweight form of error handling that improves reliability and predictability?**

**Forces.**

- *Availability*: The system as a whole should remain available, even if some subsystems or processes experience failures, since degraded functionality is better than no functionality. In case of a fault, only minimal part of the system should be affected. Recovery from failures should happen without human intervention and with minimal downtime.
- *Reliability:* Generation of incorrect outputs should be prevented, otherwise errors may propagate and the system could cause damage to the environment.
- *Safety*: If an error is detected, any functionality using the affected process should be stopped and taken to a safe state in order to prevent and minimize damages.
- *Cost-efficiency*: Design diverse fault tolerance techniques are oversized or impractical for the application, but the system needs to be able to recover from errors.
- *Real-time performance*: Control system needs to react within a certain time-limit; exceeding the time-limit causes a failure.
- *Predictability*: The system should behave in a consistent manner. If the process tries to repair its corrupted state, behavior of the system cannot be predicted, which complicates debugging and verification of reliability. Predictability includes system behavior when a fault is triggered.
- *Error handling*: Because it is impossible to foresee all possible faults, specifications do not cover all possible error situations. Error situations occur seldom, are difficult to handle and non-trivial to simulate in testing [11]. If the programmers try to implement error handling, they will make ad hoc decisions not based on the specifications (i.e. they cannot know how the error should be handled), possibly causing unwanted and undocumented behavior.

**Solution. Make processes crash-safe and fast to recover; flush corrupted state by "crashing" the process instead of writing extensive error handling code.**

Commodore 64, DOS machines and other old computers were designed to be shut down by simply turning the power off, essentially crashing the system. On the other hand, if an operating system caches disk data in memory, workstation crash may corrupt the file system, which is inconvenient and slow to repair. Control system processes and subsystems should also be designed to be easily terminated and recoverable with a simple recovery path if an error is detected, instead of guessing how error recovery should be attempted, possibly corrupting program state further and causing unpredictable behavior.

Therefore, implement error handling by terminating the process that has encountered the error. Only program extended error recovery routines if they are based on the specification or it is self-evident how the error should be handled – otherwise crash the process. However, only the module or process where the error is should be crashed, not the whole system.

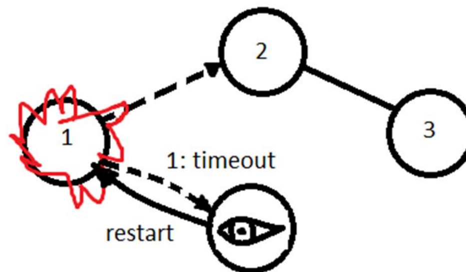



**Fig. 4.** Process 1 encounters an error and dies, after which it is restarted by the service manager, represented as an eye. If the process 2 detects a deadline overrun, it needs to stop, potentially interrupting process 3, and wait until process 1 is active again before resuming work. Alternatively the process 2 does not notice any deadline overruns and continues working normally.

Processes that have been designed with LET IT CRASH can 1) help to find faults, by making them more visible ("offensive programming"), and 2) be used to implement fault tolerance (recovery from faults or software rejuvenation). In latter case it is possible to do recovery without affecting service availability if the recovery process is fast enough. Recovery (and rejuvenation) also needs an external entity to initiate it, since the process itself has crashed (see **Fig. 4**). This pattern focuses mostly on the second case since it is more problematic to implement correctly.

You have a monitoring layer that can recover the system (e.g. by restarting). However, to detect a failure, the failed application or system service may first have to die. In this case the process terminates itself immediately upon encountering an error. Abnormal program termination can be forced e.g. by using *abort()* or *raise(SIGSEGV).* If the monitoring layer has implemented failure detection – based on watchdog, heartbeat, etc. – it can also hard-fail the service using e.g. *kill(pid, SIGTERM).*

Error recovery is performed by restarting the process. Therefore, make processes fast and easy to restart in order to minimize service failures and downtime. To keep

recovery path simple, use the single responsibility principle, thereby minimizing responsibilities of a single process. If the process encounters an error and crashes, it might be possible to recover from the error without causing deadline misses for other processes and tripping the system to a SAFE STATE. However, if a control loop has a period of e.g. 1 ms and restarting of a process that provides information for the loop takes several milliseconds, control loop execution will be interrupted temporarily.

Recovery paths can be tested extensively by terminating the system forcibly every time it needs to be shut down or restarted, instead of letting it run through a normal shutdown process. This forces the system to do a recovery during the startup

Make processes crash-safe. Processes typically handle three types of state data: dynamic, static, and internal. Internal state is related to current computations and is usually discarded after use. If a process crashes, you must think if you want to recycle its internal state. If you recycle everything you risk hitting the exact same fault again and crashing, so it might be reasonable to recycle only parts of this state. Static state is configuration data that can be easily recovered or read from other processes. Finally, the dynamic state data is generated as the program is executed by reading user inputs, interacting with other processes and environment, etc. Some of it can be computed from other data or read directly from sensors, but the critical problem is the data from user or environment that cannot be reconstructed. This data must be protected by using checkpointing, journaling or some other form of dedicated state store like databases and distributed data structures.

Implement a reporting functionality that reports failures so that they do not go unnoticed. Failure information can be forwarded e.g. by using a service manager or supervisors to send NOTIFICATION messages [12].

The corollary to the LET IT CRASH approach is that you must design your software to be ready for processes failing. There is now a possibility that a dependency is not available because it has been crashed and is being restarted. To detect this situation, add timeouts or appropriate QoS policies to interactions between components. If the timeout is triggered, move the system to a SAFE STATE. Normal operation can be resumed when dependencies are back online. A missing dependency is therefore not considered to be an error that would necessitate a crash.

**Consequences**.

+ Enables simple error handling & recovery; avoids complex error handling constructs in code, therefore improving predictability of the system.

+ Cost-effective (lightweight) form of fault tolerance that does not require use of redundancy.

+ Allows error handling to be implemented separately (externally) from the business logic, e.g. with supervisors.

+ Supports recovery from transient faults since a restart is usually enough to handle them.

+ Possible to achieve high availability (for the system as a whole, not necessary for all services provided by the system).

+ Compatible with other fault tolerant designs like redundancy.

+ Processes can be updated to new versions on-the-fly, since the old process can be killed and replaced using the normal recovery path.
+ Limits error propagation to other parts of the system (babbling idiot failure) by acting as an ERROR CONTAINMENT BARRIER [6].
+ Errors are less likely to cause the system to perform unpredictable and potentially dangerous or irreversible operations.
+ Finding faults should be easier, since they are made more visible by crashing and reporting.
- Availability of some services provided by the system is lower (when compared to redundant fault tolerance solutions) – on the other hand availability of other unrelated services provided by the system should be unaffected.
- Cannot mitigate persistent faults.
- Processes need additional code to react to missing dependencies (i.e. other services, when waiting for them to come back online).
- Possible performance cost if state needs to be saved to enable recovery.
- Recovery speed is non-deterministic since it depends on how fast the processes can be restarted, loading of saved state, loading of dependencies, system load level, etc.

**Examples.** Erlang actor model and supervisors (Erlang is used e.g. in Ericsson AXD301 ATM switches) [2]. Supervisors are processes that are responsible for starting, stopping and monitoring their child processes. The basic idea of a supervisor is that it should keep its child processes alive by restarting them when necessary [13].

**Related Patterns.** ERROR CONTAINMENT BARRIER, NOTIFICATIONS, SAFE STATE, SERVICE MANAGER, REDUNDANCY [6].

Software REJUVENATION [11][14] is a proactive technique where the system has been designed to be booted periodically. Microrebooting [11] refers to a technique where suspect components are restarted before they fail.

MINIMIZE HUMAN INTERVENTION (MHI) is about how the system can process and resolve errors automatically before they become failures [6]. LET IT FAIL could be implemented as part of MHI as a final resort or instead of MHI in case there is no specification for error handling.

## References

1. Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. Transactions on Dependable and Secure Computing, 1(1).
2. Armstrong, J. (2003). Making Reliable Distributed Systems in the Presence of Software Errors. Stockholm, Sweden: Royal Institute of Technology.
3. Dunn, W. (2002). Practical Design of Safety-Critical Computer Systems. Reliability Press.
4. Knight, J., & Leveson, N. (1986). An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming. Transactions on Software Engineering, 12, 96-109.

5. Herder, J. (2010). Building a Dependable Operating System: Fault Tolerance in MINIX 3. Netherlands: Vrije Universiteit. USENIX Association.
6. Hanmer, R. (2007). Patterns for Fault Tolerant Software. John Wiley & Sons.
7. Eloranta, V.-P., Koskinen, J., Leppänen, M., & Reijonen, V. (2010). A Pattern Language for Distributed Machine Control Systems. Tampere University of Technology, Department of Software Systems.
8. Buschmann, F., Henney, K., & Schmidt, D. (2007). Pattern Oriented Software Architecture: A Pattern Language for Distributed Computing. John Wiley & Sons.
9. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
10. EventHelix.com Inc. Manager Design Pattern. Retrieved January 2, 2013, from EventHelix: http://www.eventhelix.com/realtimemantra/ManagerDesignPattern.htm#.UOQm6kUbR8E
11. Candea, G. & Fox, A. (2003). Crash-Only Software. Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems.
12. Eloranta, V.-P. (2012). Event Notification Patterns for Distributed Machine Control Systems. Proceedings of VikingPLoP 2012 Conference. Tampere University of Technology, Department of Software Systems.
13. Erlang/OTP R16A documentation. Retrieved February 13, 2013, from: http://www.erlang.org/doc/
14. Hanmer, R. (2010). Software Rejuvenation. Proceedings of 17th Conference on Pattern Languages of Programs. ACM.

**Acknowledgements.** Authors would like to thank Robert Hanmer for providing valuable comments during the shepherding process and VikingPLoP 2013 participants for the feedback. This work was carried out under the EFDA Goal Oriented Training Programme (WP10-GOT-GOTRH) and financial support of TEKES, which are greatly acknowledged. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

## Appendix: List of Referenced Patterns

**Table 2.** Short descriptions of referenced patterns.

| Pattern | Pattern intent |
|---|---|
| BUS ABSTRACTION [7] | Nodes communicate via a message bus. The bus is abstracted so it can be changed easily. |
| ERROR CONTAINMENT BARRIER [6] | System should stop the flow of errors from one part to another by isolating them to a unit of mitigation and initiating error recovery. |
| FAULT OBSERVER [6] | Coordinate reporting to all observers that a fault is present, reported, and recovery actions escalated. |
| HEARTBEAT [6] [7] | Send a status report at regular intervals to let other parts of the system know their status. |
| LEAKY BUCKET | Implement a method to ride over transients by keeping a counter |

| | |
|---|---|
| COUNTER [6] | that is automatically decremented and incremented by errors. |
| MONITOR [10] | Support many entities of same or similar type. The MANAGER object is designed to keep track of all the entities. In many cases, the MANAGER will also route messages to individual entities. |
| MEDIATOR [9] | Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently. |
| NOTIFICATIONS [12] | Communicate noteworthy or alarming events and state changes in the system using a dedicated message type. |
| PUBLISH/SUBSCRIBE [8] | Define a change propagation infrastructure that allows publishers in a distributed application to disseminate events that convey information that may be of interest to others. Notify subscribers interested in those events whenever such information is published. |
| REDUNDANCY [6] | Maximize availability by having alternate hardware or software that can perform the same function. |
| REJUVENATION [11][14] | Periodically rejuvenate a software item by shutting it down and restarting it. |
| SAFE STATE [7] | If something potentially harmful occurs, all nodes should enter a predetermined safe state. |
| SOMEONE IN CHARGE [6] | Every fault tolerance action undertaken by the system should have a clearly identified entity controlling and monitoring the action. |
| START-UP MONITOR [7] | During start-up all devices are started in certain order and with correct delays. Additionally, care is taken that there are no malfunctions. |
| STATIC RESOURCE ALLOCATION [7] | Critical services are always available when all resources are allocated when the system starts. |
| SYSTEM MONITOR [6] | Some errors will only manifest themselves at a system level. Check for them at this level. |
| WATCHDOG [6] [7] | Build a special entity to watch over another to make sure that it is still operating well. |

# Patterns for Operating the Control System Remotely

Veli-Pekka Eloranta

Department of Pervasive Computing,
Tampere University of Technology
Tampere, Finland
{firstname.lastname}@tut.fi

**Abstract.** Modern work machine has to communicate with other systems. Furthermore, the machine operator might use the machine using remote controls to ensure the safety of the operations. These features make the system design more complex. In this paper, we present two patterns to tackle the new design challenges set by remote controlling requirements.

**Keywords:** Control Systems, Software Architecture, Patterns, Remote Access

## 1 Introduction

In this paper we present two patterns for using the machine control system remotely. These patterns are not really connected to each other and solve two different problems. The first pattern solves usability problem and the second one is related to data management and how the data on the machine can be accessed from the remote location. These patterns are a part of a larger pattern language that constitutes of approximately 80 patterns.

The patterns in this paper are presented using pattern format which is a combination of Portland form [11] and Alexandrian form [12]. First we describe the context and then present the problem in bold font face. Next we explain the forces and give the essential part of the solution in bold font face, followed by the discussion of the solution. Finally, we present the consequences and known usage separated with three stars. All patterns include a sketch of the solution and sketch icon for the pattern as well. Pattern names are written in SMALL CAPS throughout the paper.

Confidence towards patterns is marked with asterisks after the name of the pattern. Two asterisks mean that the pattern manages capture the profound problem and solution in the pattern. Patterns marked with one asterisk capture the profound problem and solution at least partially. There might be other possible solutions for the problem, but the presented one is the most likely to be the optimal solution. Patterns without asterisk may be lacking the profound solution and there might be other viable solutions. However, a solution is presented for these problems too.

# 2 Patterns

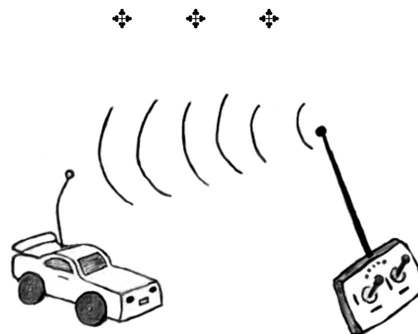
In this section, two patterns for remotely operating the machine are presented.

## 2.1 Alternative Operating Station *

a.k.a ALTERNATIVE CONTROL METHOD

…in a CONTROL SYSTEM with HUMAN-MACHINE INTERFACE the operator is typically in front of an operating station where she has controls and a graphical user interface to operate the machine. The machine itself can be quite large and thus the view from the operating station can have blind spots; there might be pillars blocking the view or the moving parts of the machine, e.g. boom, can block the view to the actual work. Sometimes the operator needs to have a better view to carry out the work tasks properly. For example, if the work implement is connected to the back of the machine and the cabin does not have rear window and mirrors, the operator would need to go out of the cabin to get better view on the task.

✣ ✣ ✣



**The operator may not be able to observe all the details of the task at hand from the default operating station or the view from the operating station is blocked, e.g. by machine itself.**

Operating the machine might require high precision control from the machine and the operator. For example, when positioning the boom for drilling, it needs to be positioned carefully into correct angle, so that the hole will be drilled to the correct direction. It might be hard to see the exact angle and the position of the boom from the cabin. Furthermore, it might be easier to control the operation from a position that is closer to the operation. For example, when lowering the tail gate of a truck, it can be hard to see from the cabin that there is nothing in the way. It is easier to move to the tail gate and see the situation from there.

Although modern work machine's cabins are designed for high visibility and comfort, there still might be obstacles, e.g. supporting pillars, blocking the view. This is troublesome when the machine is operated in a place where there is not much space around and the machine can not be positioned for better visibility. The operator could, of course, exit the cabin and observe the situation and then come back to the cabin and steer the machine and if necessary repeat this process. However, this would make the operator's work inefficient and physically exhausting if carried out over long period of time.

Safety is important aspect in the work. Usually the machine operator needs to be sure that there is no one in the working area of the machine or will enter the area while machine is working. In some cases, it is hard to see from the cabin that no one is at risk, as pillars or other machine parts create blind spots. Additionally, the machine operator herself should not be able risk her own neck by accidentally entering the working area when the work is in progress.

Sometimes the machine operator also does mechanical work and her hands might get dirty or she might be wearing work wear which prevents the usage of the sophisticated control equipment such as touch displays, etc. This might limit the range of which kind of controls can be used by the operator in this situation. Washing hands or taking protective gloves off multiple times might soon get annoying from the operator’s point of view.

**Therefore:**

**Add an alternative operating station which provides the minimal controls for carrying out the task from a position where the operator can observe the work process better.**

Design the alternative operating station so that it has only the controls, e.g. buttons, joysticks, etc., required by the special task that it is intended for. In this way, carrying out the task is efficient as there are no extraneous controls. The controls in this alternative operating station can be specifically tailored for the task, which is supposed to be carried out using the alternative operating station. In this way, the operator can wear the required equipment while using this operating station. For example, the alternative operating station’s controls can be operated with gloves on whereas the main operating station would require removing the gloves. In addition, when the alternative operating station is tailored for a specific task, it will probably not be used as a primary user interface, but only for the task it is intended for. This gives some freedom in the user interface design, for example, when considering on which screen different notifications should be shown.

It should also be considered if an additional display is needed for presenting information about the system state or for example, HMI Notifications. Furthermore, it should be decided if the alternative controls are fixed to the machine or if they are on a separate piece of equipment. The latter option can be implemented, for example, by using radio frequency controller or by using the controller that can be attached with a separate cable when the alternative operating station is needed.

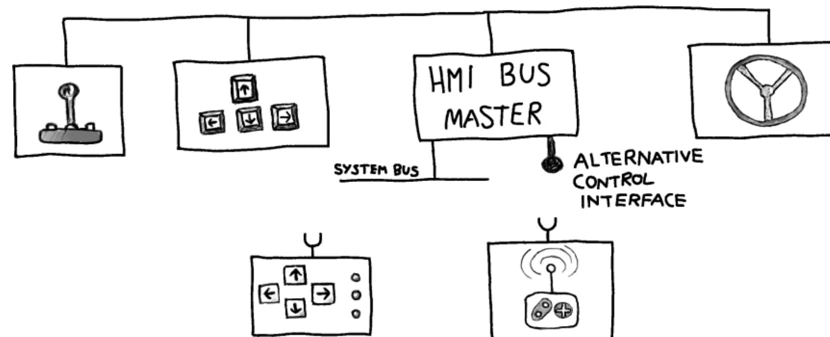



**Fig. 1.** – HMI bus master selects which control method is in use.

As shown in Fig. 1, there typically is a HMI bus master which bridges the HMI bus to system bus and delivers data and possible control signals to the engine, transmission, etc. using the system bus. So, it is natural to define and implement an interface to HMI bus master which provides necessary methods for alternative operating station. Now, when using the alternative operating station, the HMI bus master can decide which control signals to use: the signals coming from the alternative operating station which is attached to the interface or from the primary HMI which is attached to the HMI bus. The HMI bus master must ignore the command messages coming from the source that is not selected to be used, except emergency stop that should be functional. This is also stated in the European Union's machinery directive, section 1.2.5 [10]. Furthermore, the switching to the alternative operating stations must not cause unintentional movements in any case. For example, if there is another operator pressing a button while the switchover is being carried out, it should not cause any movements.

In some cases, where the alternative operating station is fixed to the machine, a separate bus for the alternative operating station is used. If this is the case, alternative control interface in Fig. 1 is replaced with another bus. HMI bus master then selects from which bus, the received control commands are bridged to the system bus and to controllers of the system.

The control method in use should be selected from the HMI in the operating station in the cabin of the machine, e.g. by using a switch or button. In this way, the control system can decide which safety mechanisms should be used and e.g. limit boom movements accordingly. Different set of safety precautions, e.g. which of the functionalities can be used, should be applied when using the alternative operating station, as the machine operator is likely to be closer to the work implement when using the alternative operating station. For example, if a forest harvester is operated using the remote control, there is a risk that the operator can hit herself with the log while maneuvering the boom.

If the OPERATING MODES pattern has been applied, it is rather easy to implement a separate alternative operating mode for the machine. The nodes of the system can change their state to the corresponding mode when the switch selecting the alternative

operating station in use is turned to different position. For example, nodes in the HMI bus can stop sending control messages when they enter an alternative control mode. Furthermore, the nodes on the system bus can start acting differently, e.g. actuators can limit the range of movements. The operator activates the alternative control mode from the switch in the cabin and after that the alternative control station can be used. The deactivation of the control method is carried out in similar way. Other strategies for changing the mode can be used as well, for example, the control method is automatically switched to the normal mode when the cable of the alternative control unit is unplugged.

Typically, the alternative operating station is used in mobile work machines. However, it can be used also for example in process automation systems, where the monitoring station might be physically far away from the actual process, so it might be impossible to see the process from the monitoring station and control the process, e.g. in fault situation. So, it might be advisable to place simple controls for manually operating the machine near the places where they might be needed to sort out the problems. For example, in paper mill, the paper flow might get interrupted because of a fault and the continuous sheet of paper might break and it has to be fed through the system before starting the automated paper flow. So, there might be controls for manual feeding right next to the places where the paper roll is probable to break.

The solution presented in this pattern is not suitable for implementing the emergency stop switch. It is usually implemented on the hardware level to make sure it is functional even though there is software fault. Hardware implementation also gives usually faster response times. Additionally, one should remember that even when the alternative operating station is used, the emergency stop button should be functional from any operating station.

The VARIABLE MANAGER pattern helps to share the data for the alternative operating station. If HMI Notifications pattern has been applied to show the machine operator information about the events occurring in the system, it needs to be decided on which display the notification is shown - probably on the one that the operator is currently using. If alternative operating station does not have its own display to inform the operator BEACON pattern could be applied to draw operator's attention to machine's primary controls when a noteworthy event occurs.

If the alternative operating station is not fixed to the machine, the connection to the control unit may be lost. For example, the battery of the control unit can be depleted or the cable may break. There should be a mechanism to detect these kinds of situations so that the machine can enter SAFE STATE if the connection to the alternative control unit is lost. This is also pointed out in the machinery directive section 3.3.3. [10]. Typically, if the connection is lost, a separate emergency stop mechanism is used to stop the machine. For example, if the system has remote control unit, it can have CAN module which sends the commands to CAN bus. In addition, the remote control unit has safety certified relay telling if the connection is ok. This relay breaker is connected to emergency stop circuit, so it stops the machine when necessary. In this way, the emergency stop messages are not delivered through the alternative operating station's interface. HEARTBEAT can be used to monitor the health of the connection between the HMI bus master and the alternative control unit. Furthermore, when us-

ing remote control unit as an alternative operating station, one should take care of security, so that the system can not be controlled from the unauthorized alternative operating station.

The operator can control the machine or work implement near the place where the actual work is done. Now, the operator has a good view on the work she is carrying out as obstacles are not blocking the view. Still, the system should be designed so that the need to go outside the control station should be minimized.

Alternative Operating Station functions also as a redundant control station which can be if the main control station is broken down. This makes it possible to do some controlling in fault situation, for example, move the boom to the transportation position.

The connection to alternative control unit may be more unreliable and it must be monitored. This may need some additional resources, e.g. bandwidth, CPU time, etc.

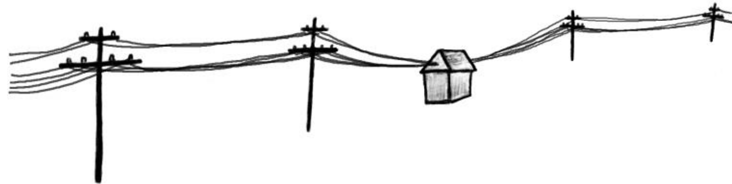
As the machine operator is controlling the functionality close to the work implement, it potentially puts the machine operator in danger. So care must be taken while operating the machine from the secondary operating station, the machine operator can not end up in the way of the boom or other moving parts of the machine.

Alternative Control Station increases the development costs and might increase the cost of the machine. Therefore one might consider providing it as Control System Option as the customer may not be willing to pay for the extra control station.

Ground heat wells are typically drilled with low-end drilling machines. The machine usually needs to be driven to the drilling site in narrow pathways of the garden. The machine operator has to avoid destroying the vegetation or hitting garden sheds or other obstacles. The cabin of the machine has poor visibility to the front side of the machine as there is boom blocking the view and if the machine operator was to drive the machine from cabin she would be likely to hit some of the obstacles. Thus, the machine operator is provided an alternative control method: remote control unit. The operator turns the alternative control method on from the HMI in the cabin by pressing a button. After this mode change, the machine responds only to steer commands from the remote control unit. Now the machine operator can go in front of the drilling machine and steer the machine with better view. On the architecture level, HMI bus master is set to alternative control mode, where it ignores all the control messages coming from the HMI bus. Instead of it, the alternative control interface is used to receive the control commands which are then sent forward to the boom and drilling controllers. Furthermore, HMI bus master sends status information to the remote control unit using remote controller's interface instead of bridging this information to HMI bus and to actual devices.

## 2.2 Remote Access

...there is a distributed CONTROL SYSTEM and the system has a MESSAGE BUS that is used for communication between nodes. However, all the data the nodes have to access may not be stored locally on-board the work machine. For example, in the case of drilling machine, drilling plans are produced remotely by managers in a mine control room and the FLEET MANAGEMENT application on the machine needs to access them. In addition, some third-party applications may need to access remote data sources, e.g. navigation software may need to download map updates. Similarly, the remote systems might want to access the data on the machine, e.g. remote diagnostics application would like to read diagnostic data from the machine.



**All services using the data that the machine collects do not necessarily reside on-board the machine. Similarly, the applications on-board needs data which is produced off-board in a different location.**

In a ubiquitous environment, the applications inherently need to communicate with other applications and systems. Applications need to exchange information in order to produce additional value. For example, if the application is meant to display drilling plans, the application is not useful for the user if the application can not access the newest drilling plans created by explosives expert in the mine control room. In general, work planning applications are typically located in a remote place, so applications on-board the machine needs a way to retrieve data from them. On the other hand, applications from the remote locations may use the data produced by the machine to produce additional value.

DIAGNOSTICS data should be transferred to maintenance service's system so that the maintenance personnel can analyze the data before the scheduled maintenance break and find out if spare parts need to be ordered from the manufacturer. However, the machine might be far away from the service station, so it is not feasible to fetch the DIAGNOSTICS data with USB stick or some other physical media. Especially, if there are many machines and some of them are located abroad, the data gathering is hard.

The controllers of the machine may have limited processing power and thus detailed analysis of DIAGNOSTIC data can not be executed on-board. Furthermore the central repository can store data from a longer time period than could be stored on the machine as the machine typically has limited storage space.

During the life cycle of the machine, the control system software is likely to be updated (see UPDATEABLE SOFTWARE pattern). The new software version could offer higher productivity or improved features. However, updating each machine separately would be too laborious for maintenance personnel. If the update packages are delivered to the customers with USB sticks or other media, they might feel that service quality is low as the machines are not updated by the manufacturer. Furthermore, the customer might have multiple machines and it would also be too laborious for the customer to distribute the updates.

If production plans change, the machine operator needs to be informed about the changes as soon as possible to avoid working on tasks which results are not needed anymore. On the other hand, the production planning system or FLEET MANAGEMENT might need to know already carried out tasks, e.g. how many cubic meters of pulpwood is already sawed today. If a machine breaks down, information about already completed work could help FLEET MANAGEMENT system to re-allocate tasks allocated for the machine to other machines. This kind of real time planning and reporting needs frequent update cycle of work plans.

Sometimes the work environment where the machine is used is such that it would be more comfortable or safer for the machine operator to operate the machine remotely. For example, demining is safer to be done remotely.

**Therefore:**

**Add a remote connection gateway on-board which enables communication between the machine and the remote party. The remote connection gateway transforms the used messaging scheme to suit the local and remote parties' needs and can take care of authentication.**

In the simplest case the remote connection gateway is a node attached to the bus bridging the traffic from and to the remote location. For example, the gateway node reads CAN messages that are targeted to the node and sends the data to remote location over TCP/IP. The other way round, the incoming messages are converted to CAN messages and sent to the bus. For many bus technologies, COTS (Commercial Off The Shelf) solutions exist for implementing this kind of remote connection gateways. The remote connection can be seen as a special case of MESSAGE BUS.

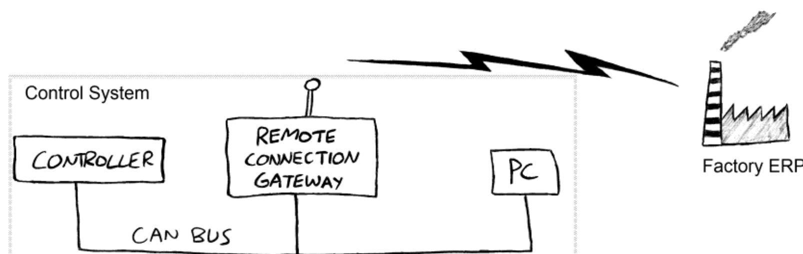



**Fig. 2.** Remote connection gateway connected directly to CAN bus.

If the SEPARATE REAL-TIME pattern has been applied the system is divided into the real-time machine control level and non-real-time operator level. The operator level is typically implemented with a PC. If this is the case, the remote connection gateway has s a natural place on this PC as the operating system offers ready-made facilities, e.g. communication interface to manage communication with remote parties. If a communication interface exists, applications can use it in the operating system's native way. Furthermore, as remote connection gateway is placed on the operator level, it can not interfere with real-time machine control.

One should decide if the machine's control system which connects to remote party and acts only as a client or does it need to support incoming connections as well, and act as a server. If this is the case, there needs to be a service taking care of incoming connections. Typically this server is deployed on the PC. This means that the server resides on the operator level, and will not interfere with the real-time machine controlling. Sometimes this server can be just an in-house application listening to a certain TCP/IP port or sometimes a ready-made implementation such as Nginx [6], lighttpd [7] or even Apache web server [1]. If file server is needed, one could use Cerberus [2] for instance.

A server creates an attack vector for hackers and thus one should carefully consider deploying one. The server's software might have vulnerabilities that can be utilized to gain access to the machine. In addition, adding a server makes the machine vulnerable to DoS (Denial of Service) attacks meaning that it becomes unavailable to its intended users. One common method of DoS attack is to saturate the target server with external communications requests, so much so that it cannot respond to legitimate traffic, or responds so slowly that it will be rendered essentially useless. In general, one should make sure that proper measures are taken to ensure the security of the system. One should also consider if the communication needs to be encrypted to prevent eavesdropping the traffic. If so, one might want to use VPN (Virtual Private Network) [3] technique, e.g. IPSec [5], OpenVPN [9], or Secure Shell (SSH) [4] to make the connection secure.

The remote connection gateway can be used to transfer the DIAGNOSTICS data from the machine to the manufacturer or to the service station. Data can be transferred automatically without requiring the machine operator to do anything. The remote party might be a cloud service, so it has enough capacity to process the data from a plethora of machines. Now as the data is transferred to this kind of environment with potentially unlimited processing capabilities more thorough analysis on the data can be run. Furthermore, as data from multiple machines is gathered in the centralized place, data originating from different machines can be compared with each other to detect different patterns and malfunctions. For example, gathered data of oil pressure readings can form a certain pattern in normal use. Now, if the machine does not comply with this pattern, it can be a malfunction that needs to be inspected more closely. Over a longer period of time, data from different malfunctions can be also gathered and recognized which kind of malfunction causes certain changes in the patterns of normal behavior of the machine.

Depending on the available communication channel between the machine and remote party, the properties of the channel may vary. For example, sometimes the

communication may be expensive and the transferred data amount should be limited. One should refer to DYNAMIC MESSAGE CHANNEL SELECTOR pattern on how to choose the optimal communication channel for each situation.

If the VARIABLE MANAGER pattern has been applied to share the system wide information as variables, the REMOTE ACCESS can be used to share this information to remote location. If the machine is working as a part of a fleet, the REMOTE ACCESS is a key component in coordination between multiple work machines. In this case, the SUBSYSTEM ADAPTER pattern may need to be used to ensure compatibility with other, possibly legacy, systems. MACHINE-TO-MACHINE COMMUNICATION introduces peer-to-peer communication between work machines. In this case the REMOTE ACCESS is used to communicate with other machines. When implementing a remote access to a work machine one must consider the implications for the functional safety. There is a systematic approach to assess whether a given technological solution for remote access to control system implies an unacceptable risk in, in terms of jeopardizing the safety integrity level (SIL) of the system [8].

As the information on the machine can be accessed remotely, the data gathering for preventive maintenance, production management, etc. is faster as the data can be transferred more frequently. Additionally, the data transfer becomes independent of the machine location as the machine does not have to be accessed physically.

Software updates (see UPDATEABLE SOFTWARE) for the control system can be delivered over the air. This enables faster delivery of the updates and decreases the updating effort from the machine owner as the updates do not need to be delivered using removable media, e.g. USB memory sticks. For updating the software there are two options: pull or push. Regardless of which approach is selected, the updates should be installed only when the machine operator (or service person) wants to do so. Furthermore, updating system requires taking it to the special updating mode (see OPERATING MODES). Furthermore, as the remote connection gateway is the only access point to the machine, if vulnerabilities are discovered, only this component needs to be updated. On the other hand, when new vulnerabilities are discovered, the software needs to be updated more urgently to avoid the consequences of a possible attack. So REMOTE ACCESS increases the required update frequency.

If the customer thinks that the machine is malfunctioning, the maintenance personnel can take a remote connection to the machine and try to diagnose the problem without physically visiting the machine. This might reduce costs as the maintenance persons do not need to travel the remote location. On the other hand, if the malfunction can be diagnosed remotely and it can be recognized if there is a need to bring the machine in for maintenance or not, it would reduce costs. Additionally, if the system has Parameters, the remote connection can be used to adjust these parameters, so calibrating the devices on-board becomes easier as the maintenance person does not need to visit the machine physically in order to do such adjustments.

Production reports and work orders can be transferred more frequently. This makes FLEET MANAGEMENT more flexible.

Possibility for REMOTE ACCESS creates risks for unauthorized access to the machines information. This might also be a reason why one might not want to implement remote controlling of the machine using this approach. Furthermore, denial-of-service attacks could slow down the node where the server is running. The remote connection gateway requires some processing power (especially in a case of a server), so it might slightly decrease the performance of the operator level node in general.

As the remote communication is isolated in one module, it is easy to limit the bandwidth usage, e.g. communication is possible only for trusted applications.

Remote connection gateway makes it possible to implement remote operating of the machine.

Company manufacturing excavator discovers a serious software bug in one of their control system versions. Unfortunately, they don't have records of machines and which software version they have. However, REMOTE ACCESS pattern has been applied and the control system has a server that can be connected from the factory when the machine is powered. Manufacturer implements a client application which connects to the work machine and checks the software version. If the control system has the faulty software version, the manufacturer can send the software update package to the machine owner, so they can update it.

## 3 Acknowledgements

The author would like to thank all the participants of the VikingPLoP 2013 for valuable feedback on this work. Without you the patterns wouldn’t have improved to the level where they are now. Additionally, I would like to thank Marko Leppänen for shepherding this paper.

## 4 References

1. The apache software foundation: The apache web server. (2013). Website, available online http://httpd.apache.org/, visited 19.9.2013
2. Cerberus FTP Server (2013). Website, available online http://www.cerberusftp.com , visited 19.9. 2013
3. Mason, A. G. *Cisco Secure Virtual Private Network*. Cisco Press, 2002, p.7.
4. IETF Network Working Group: RFC 4252, *The Secure Shell (SSH) Authentication Protocol*, January 2006
5. Kent, S.; Atkinson, R.: *IP Encapsulating Security Payload (ESP)*, RFC 2406, IETF, November 1998

6. Engine X (nginx) HTTP server. Website, available online http://nginx.org/, visited 19.9. 2013
7. Lighttpd webserver. Website, available online http://www.lighttpd.net/ , visited 19.9. 2013
8. Jaatun, M. G.; Grøtan, T. O.; Line, M.B.: Secure Safety: Secure Remote Access to Critical Safety Systems in Offshore Installations. In proceedings of 5th International Conference ATC: Autonomic and Trusted Computing, Editors: Rong, C.; Jaatun, M.; Sandnes, F.;Yang, L. and Ma, J., LNCS, Volume 5060, 2008, pp. 121-133, ISBN; 978-3-540-69294-2
9. Open source VPN (OpenVPN). Website, available online http://openvpn.net/, visited 19.9. 2013
10. Directive 2006/42/EC of the European parliament and of the council on machinery, and amending. Directive 95/16/EC (recast, May 2006). Available online, http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2006:157:0024:0086:EN:PDF, visited 19.9. 2013
11. Portland Pattern Repository. 2003. Portland form. http://c2.com/cgi/wiki?PortlandForm, retrieved 19.9. 2013.
12. Portland Pattern Repository. 2011. Alexandrian form. http://c2.com/cgi/wiki?AlexandrianForm, retrieved 19.9. 2013.

# Patterns for Decoupling Hardware and Software

Johannes Koskinen

Department of Pervasive Computing
Tampere University of Technology
Finland

{firstname.lastname}@tut.fi

## 1 Introduction

The patterns presented in this paper are part of a larger pattern language that is currently formed in collaboration with large global machine control companies. The patterns have been collected from the real-life systems using architecture evaluations and interviews. The previous version of the language is available in [1].

A control system is a device, or set of devices to manage, command, direct or regulate the behavior of other devices or system1. In this paper by an embedded control system we mean a software system that controls large machines such as harvesters and mining trucks. Such systems are often tightly coupled with their environment. For example in case of a harvester, harvester head hardware needs special-purpose applications to control it. In a distributed control system, the system is divided into subsystems with each controlled by one or more controllers. Networks connect these controllers.

## 2 Patterns

In this section a set of patterns from the pattern language (refer **Fig. 1**) is presented. The selected patterns for the paper are HARDWARE ABSTRACTION LAYER, OPERATING SYSTEM ABSTRACTION and VIRTUAL RUNTIME ENVIRONMENT. The pattern language graph could be seen as a designer's map for solving design problems. The design process begins so that the first pattern to be considered is CONTROL SYSTEM in the middle of the graph. After the designer has made the design decision to use the pattern, she may follow the arrows to the next patterns. An arrow means that the following pattern can be applied in the context of the resulting design from the old pattern. In other words, the subsequent pattern refines the design. However, a single pattern may be used regardless of the usage of previous patterns if the context of the pattern matches the current design situation.

The patlets for the patterns as well as the referenced patterns not included to this paper are presented in Table 1.

[1] http://en.wikipedia.org/wiki/Control_system

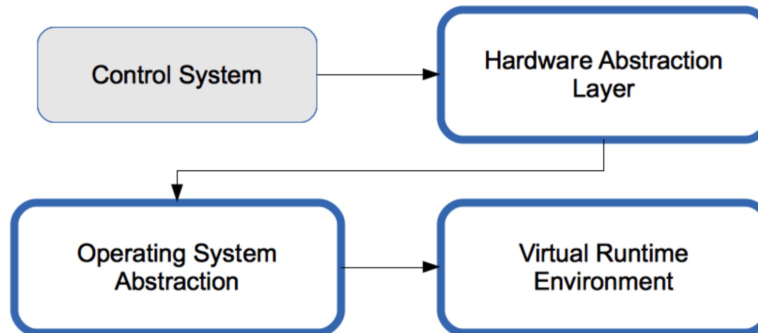



**Fig. 1.** Part of the pattern language for distributed control systems.

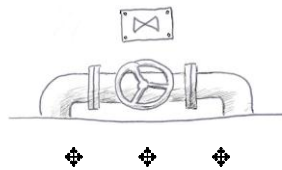
**Table 1.** Patlets for the included and referenced patterns.

| Pattern | Patlet |
|---|---|
| CONTROL SYSTEM | **Productivity of a work machine can not be increased significantly anymore only with traditional way of building the machine, i.e. using hydraulics, electronics and mechanics. Additional ways to control the machine needs to be introduced in order to enhance the system functionality and to increase the amount of automated functionalities.** <br> **Therefore:** <br> **Implement control system software that observers the work environment and controls the machine accordingly. Control system software can automate many functionalities which have previously been carried out by the machine operator manually. In addition, the control system software can provide the operator more fine-grained information about the work environment.** |

| | |
|---|---|
| THIRD-PARTY SANDBOX | **The end users require more features to the system and the manufacturer has to outsource the development of some applications that are not in their core competence area. These applications cannot always be trusted not to compromise the operation of the system as a whole inadvertently or on purpose.**<br><br>**Therefore:**<br><br>**Provide an interface and tools for third-party application developers. Third-party applications can use the machine services only through this interface so that they will not interfere with the machine's own applications. The interface provides common ways to access data and services.** |

## 2.1 Hardware Abstraction Layer

...there is a Control System where applications must access two kinds of devices to control the machine: sensors and actuators. Sensors provide input data for the applications and applications use actuators, e.g. valves and motors, to manipulate the environment. It is usually possible to identify different types of devices. For example, all the temperature sensors have common characteristics: they output an analog or digital value according the current temperature measurement. In addition, the possible actions for the all devices in the same category are more or less the same: one can read a value corresponding the sensor's measurement or give control signals for the actuators. However, even though the devices were meant to be used for the same purpose, the outputs and control signals might vary between device models and devices from the different vendors.



✥ ✥ ✥

**Each vendor may have its own way to control hardware devices. If all the devices are controlled in vendor specific way, it makes the application code to dependent on the selected hardware. To make applications portable, the hardware should be decoupled from the applications.**

The life cycle of the control system is usually long and it is independent from the life cycle of its devices. In many cases, the devices' life cycle is shorter. During the life cycle of the product, there may be a need to change the device vendor, for example, if the current device is not available anymore. There seldom exist standardized ways for different vendors' devices to interpret the control signals. Thus, the change of device vendor may usually require changes to way the hardware is controlled. For example, each vendor may have different method for reading the measurement of the temperature sensor. One vendor could prefer 4-20 mA current loop, that corresponds to certain, usually vendor specific temperature range the other outputs the reading digitally. The change of the device may require changes in the application code. It may be error-prone and expensive to modify code.

The system may have Control System Options and there might arise need for new options. So, it might be difficult to know all the needs for future changes in hardware. New devices may be required or some of the existing ones are discarded as the needs change. Moreover, there may even emerge a need for different sensor types. Still, the application should be unaffected by the implementation details of the devices. For example, changing the vendor of the sensor should not cause changes in various places in the application.

In a product platform, there can be multiple machines of the same product family. These machines have slightly differing hardware setup. To save costs, the same control applications may be used in various products, regardless of the physical devices. Thus, the application should be easily portable.

There often are vendor-specific features in devices. Usually, using those features would make application development faster, but it would make the application depend on the vendor-specific hardware as these features are not available on other vendors' products. To avoid vendor lock-in, it should be possible to have a common feature set that is used for all the products.

**Therefore:**

**Create a Hardware Abstraction Layer (HAL) between the application software and the hardware implementation of the controlling mechanisms of the devices. In this layer, provide generic interfaces to access the devices of a certain type in a uniform way. HAL abstracts implementation details of the hardware under these interfaces.**

Classify the devices into groups by their functionality. For example, temperature sensors form a single group regardless of the actual hardware implementation used for the temperature measurement. In a similar way, all bus interface cards are classified into the same group. For each classification group, create a generic interface to HAL, called application interface, with methods to control the devices in the group in a unified way. The implementation details of the hardware are hidden by the interface. The control application uses the application interface to command the device instead of accessing the device directly. In practice, this means that programmers don't need to know the details of individual devices, and the applications will be compatible with any device in the group. This makes the applications portable as long as the interfaces of HAL remain the same.

HAL translates the method calls from the applications to device dependent control signals. The application can read the measurement data from the sensors in common format of the device type using methods provided by HAL. Such formats can be simple binary (on/off) or a sample of continuous signal. In the latter case, the data can be expressed, for example, as a percentage of the device's maximum value, absolute turning angle in degrees, or in predefined types like in degrees of Celsius. As the measure data can be in various formats, VARIABLE VALUE TRANSLATOR can be used to translate these types from one format to the other. In addition, to increase portability, HAL can provide generic data types for the applications. For example, if the application needs to read a temperature sensor's value, it simply uses reading method available in the application interface to get the value from the sensor. There are several types of sensors which have differing word lengths for the return value. Thus, it is abstracted with the generic data type which is processor architecture independent. The layer can also contain generic methods, such as self-checking, initialization and configuration of the devices that are common among all the devices.

The device vendors can usually provide device configuration sheet file (EDS, Electronic Data Sheet) which are used to configure the certain device or device types. The file can also be used to configure the application interface methods of HAL. Format of EDS file may depend on the vendor, development tools and environment used, and sometimes the configuration must be done manually. HAL can sometimes provide a common interface for the developers to implement devices compatible with HAL.

Some device types may already have generic, standardized interface and all devices implementing the interface can be accessed in the same way.

As there usually are variations in capabilities of the devices inside one device group (like resolution of different sensors), the HAL interface design is usually based on the least common denominator. However, the interfaces can overcome the limitation by containing support for reflection, like querying the type, and information on the capabilities of the device. The application can use the information to adapt its PARAMETERS according to the hardware available. However, this kind of reflection reveals information on the details of the hardware to application and can cause abstraction leaks. An abstraction leak makes the software porting to other hardware platforms more difficult [LEAK]. If vendor-specific properties of a device are used, it usually leads to vendor lock-in as it might be difficult to find a replacement part with the same properties. Thus, it is usually worthwhile to use only common capabilities available in all the devices. **Fig. 2** illustrates HAL layer between the application and the hardware as well as the usage of drivers.

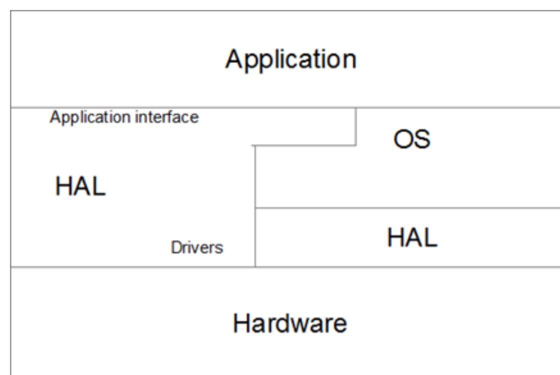



**Fig. 2.** Illustration of an architecture having two HALs: One for applications and one within the operating system (OS)

As HAL adds an extra layer between hardware and the applications, it increases latencies and may jeopardize determinism of the system. Thus, it may be difficult to make real-time applications with strict real-time requirements. In addition, as it abstracts the actual actions needed to access the device, the required time for the access is usually unknown for the application developer. For example, reading a value from the sensor may be shown as a single method call for the developer, but in practice it might be time consuming operation requiring a lot of communication between the controller and the sensor.

If a controller contains an operating system, the operating system uses its own HAL. The idea of HAL in operating systems is to abstract actual hardware from the kernel of the operating system, thus making the operating system more portable (see **Fig. 2**). In addition, the application can still use its own HAL to access the hardware. To abstract operating system from the application, one can use OPERATING SYSTEM ABSTRACTION.

If VARIABLE MANAGER has been applied in the system, the variable manager provides variables that abstract details of the devices in similar way as HAL. VARIABLE MANAGER can use HAL to update its variables to avoid device dependency. MESSAGE BUS decouples different hardware devices that are connected directly to CAN bus, such as nodes and CANopen sensors. The bus provides common messages and standardized interface to access the devices so there is no need to use HAL for those devices. Sometimes, for example with FLEET MANAGEMENT, there is a need to abstract the whole machine. SUBSYSTEM ADAPTER can be applied to tackle this problem.

Peng and Dömer introduced unified hardware abstraction layer architecture for embedded systems in [PD]. In [MC] McCollum describes how TYPE LAUNDERING pattern can be used to abstract hardware interfaces from application layer code. HAL pattern can be seen as a layer introduced by LAYERS pattern [POSA4] or LAYERED ARCHITECTURE [PLOPD1].

As all device dependent operations are encapsulated to HAL, a device can be replaced with the new one without any changes to the applications. However, it may be difficult to design an interface that is generic enough, but still takes into account the differences between the devices of the same group. In addition, as the interfaces are designed based on the least common denominator, some advanced functions of the device may be unavailable for the application developer. However, this helps forcing the developer to use only commonly available features, thus decreasing the possibility for vendor lock-in.

Similar hardware devices or hardware devices of the same type can be used through a uniform interface. This makes application porting easier. Moreover, programmers of the applications don't need to care about implementation details of the individual devices. However, implementing the interface and device drivers may require more effort than accessing the devices directly. In addition, without proper support in HAL or existing device drivers, the device cannot be used by the application or developers may try to circumvent the abstraction layer.

It may be difficult to make applications using HAL if they have strict real-time requirements. HAL abstracts the actual actions needed to access the device so the required time for the actions are usually unknown for the application developer. In addition, HAL adds an extra layer between application and devices, so using it may increase latencies and require more processing power.

In a harvester, a sensor is used to read oil temperature for the engine control application. There are several vendors providing the temperature sensors. However, the properties of the sensors differ slightly between vendors. The sensors are directly connected to boom controller's I/O-ports with 4-20mA current loop connection, so there is no bus to decouple the device from the controller. The measurement area may differ between sensors so that in one sensor 4mA could mean -20C while the other sensor has 0C as the lowest possible measured value. The harvester manufacturer

does not want to commit to one sensor provider for financial reasons. The manufacturer invites various vendors to tender for subcontracting of sensors yearly in order to get the best offer. As modifications to application code are expensive and error-prone, HAL is used to hide the differences between sensor models. When the application needs the temperature value, HAL is used to scale and linearize the sensor's raw temperature value to 0..100C according to characteristics of the actual sensor used. This also allows code reuse since the same controlling algorithm can be used with various work machine hardware if the same HAL is available.

[PD] Hao Peng, Rainer Dömer, "Towards A Unified Hardware Abstraction Layer Architecture for Embedded Systems," Center for Embedded Computer Systems, Technical Report 12-14, November 2012.
[MC] Cliff Michael McCollum, Type Laundering as a Software Design Pattern for Creating Hardware Abstraction Layers in C++, University of Victoria, 1996
[LEAK] Kiczales, Gregor (1992). "Towards a New Model of Abstraction in the Engineering of Software

## 2.2 Operating System Abstraction

...there is a CONTROL SYSTEM where hardware of the controller is abstracted by applying the HARDWARE ABSTRACTION LAYER pattern. It provides a hardware independent layer so that the applications can access various hardware components in the same way. An operating system is used to provide common services, such as device management, scheduling and memory management for the platform. In some cases, the applications may have different life cycle than operating systems used for the platform. For example, support for the current operating system is ended and the operating system must be updated in the middle of the product's life cycle. If the operating system is changed, the application needs to be ported to the new operating system.

**The life cycles of the applications and underlying operating systems may differ. Still, it should be possible to change the operating system with only minimal modifications to the application code.**

Operating systems provide useful services for the applications and abstract the hardware from the application. The operating system provides an API, which the application uses to access hardware. Usually these APIs are unique for the operating system and porting an application from one operating system to the other may be

challenging and porting should not affect to the control logic of the application in different platforms.

The operating system may be a freely available OS such as Linux, a commercial off-the-self product (COTS), or an in-house product. Usually the same operating system is selected to be used for all the products in the product family.

In some cases, the customer may require a specific operating system to be used in the work machine, to ensure the compatibility of the machine with customer's other information systems. In addition, the customer may want to be able to use certain 3rd party applications requiring a specific operating system. To support these requirements, the control system applications need to be compatible with various operating systems.

As the operating system abstracts the hardware, the applications can be used in various products. However, the applications depend on the operating system used and when the operating system changes, the application needs to be adapted for the new operating system. Still, the same application is used for various product platforms. In addition, the number of different versions of the same application should be minimized as the supporting and updating the versions is error-prone and usually requires extra effort.

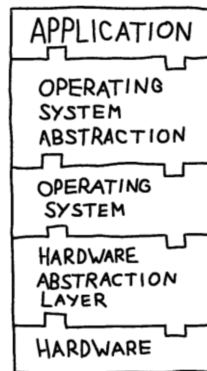
Operating system version probably needs to be changed when the support for the current version is ended and the operating system version has become obsolete. In some cases, the application interface of the operating system changes as the version changes. This may require modifications to the application.

There may be legacy applications that run on top of the operating system that is not available anymore. Still, it should be possible to port these applications to a new operating system with little effort as developing the new version of the application may be costly.

The application cannot be easily changed or replaced with new one as the new application with only few usage hours may contain errors that are not yet found. In addition, maturity of the applications is an important aspect especially in safety critical environments. Safety standards (such as IEC61508) require revalidation of the application after modifications and thus it is desirable to change software as little as possible.

**Therefore:**

**Create an abstraction layer, which implements all OS dependent services. In the application code, use only this abstraction layer for the services.**




Identify all OS dependent services, such as memory management and graphical UI support, needed by the application. Create a new, operating system dependent layer called Operating System Abstraction (OSA) to satisfy the needs and use its interface instead of ones provided by the operating system. In this way, it is possible to have stable interface for the applications even though the interfaces of the operating system changes. The abstraction layer should contain the services that the operating system would normally offer to the applications. The application calls the abstraction layer, which in turn translates a call to one or more corresponding operating system calls. Usually, OSA includes various operating system independent data types for the applications.

Operating systems have a different set of services that they provide for the applications. In some cases, it might be very difficult or even impossible to provide a service in OSA layer if the underlying operating system has no support for that. For example, memory management system for dynamic memory allocation can be implemented by OSA layer regardless of the support from the operating system. However, if the operating system does not support memory protection, it may be impossible to provide that kind of support by the abstraction layer.

Some operating systems may require additional steps, like registering the application or calling initialization method before the application can be executed. To ensure that the methods are executed when necessary, the application should always call the initialization method of the OSA when starting the application - even if it is not required by the current operating system.

Operating systems may already have a common, standard interface, which can be used instead of OSA. For example, POSIX (**P**ortable **O**perating **S**ystem **I**nterface) [IEEE1003] is a specification defined by the IEEE for maintaining compatibility between operating systems. Using the APIs defined by POSIX, it is possible to change from one operating system to the other as long as both of the systems support POSIX interfaces. It is usually possible to port the application in the new system just by recompiling it.

For some operating systems, there are existing cross-platform application frameworks, such as Qt [Qt] or SDL (Simple DirectMedia Layer) [SDL] which can be used to abstract the underlying operating system. With these frameworks the application can be ported to use other operating systems just by selecting corresponding library implementation for the target operating system and recompiling the application.

Furthermore, there are emulator libraries available for some operating systems. These libraries provides a compatibility layer which acts like an abstraction layer so that the application can be designed for one operating system, but compiled also for another OS. The compatibility layer translates the operating system calls like OSA. Unlike OSA, the compatibility layer is not linked with the application, if the application is compiled for the "native" operating system. With similar technique, the whole operating system can be emulated with an external emulator so that the same application can be executed on various operating systems. The emulator consists of a compatibility layer and a method to start the application. The emulator starts the application so that it will use the compatibility layer for operating system method calls. The application itself does not know whether it is run with native or emulated operating system.

If the hardware is likely to change and the recompilation of the application is not possible, one could apply the VIRTUAL RUNTIME ENVIRONMENT pattern to virtualize the whole runtime environment including hardware of the system and the operating system. In addition, the application needs no recompilation even if the application is used in various platforms.

OSA encapsulates all the OS dependent parts; a single team can take care of developing OSA layer for various platforms. In addition, the common API makes the application development easier as the developers need not to know all characteristics of various operating systems. However, developing OSA requires a lot of expertise.

Using generic data types instead of the ones provided by the programming language eases porting an application from one hardware platform to the other as the compiler could take care at least some part of the required adaptations. For example, integer types with well-defined bit size may prevent compatibility problems when the processor of the control system changes.

The application can be executed with various operating systems as the application does not depend on the operating system used. If the operating system changes, the application code does not need to be changed. However, OSA layer needs to be updated when the abstracted operating system changes. In addition, the layer and the application need still thorough testing after changes in the operating system.

Usually the cross-platform application frameworks focus only on certain area (like graphical user interface), so it might not have all the functionality required by the application. In that case, some of the functionality must be provided by using other means, like using other frameworks or writing own library for the functionality. However, it may be impossible to support required functionality as it may require support from the underlying operating system. As the services provided by the OSA layer are

common with all the operating systems, supporting various operating systems may prevent using most advanced features of other operating systems. Those features might be more efficient or provide some extra functionality.

Using an extra layer between the application and the operating system usually decreases performance of the system. This could be critical in applications with strict real-time requirements.

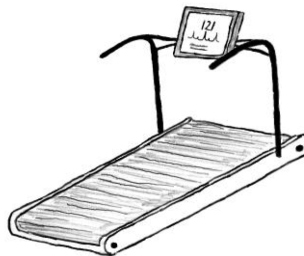
If cross-platform application frameworks are used, the application becomes dependent on the libraries and their changes. Moreover, using some proprietary libraries can lead to vendor lock-in.

In the harvester's cabin, there is a PC providing graphical user interface for the operator. The hardware of the PC may vary in different harvester products. The harvester vendor has previously selected Windows operating system for the product family of the harvesters, but now the customer wants to use Linux operating system in the cabin PC. As the application for user interface is designed to be operated with various operating systems, it uses Qt to implement various UI elements. In this way, it is easier to develop the application so that it can be executed on different harvester products - just by using the corresponding Qt library and recompiling the application.

[IEEE1003] IEEE 1003.1-2008 - IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))
[Qt] http://qt-project.org/
[SDL] http://www.libsdl.org/

## 2.3 Virtual Runtime Environment

……there is a Control System with one or more controllers which execute one or more applications each. These control applications provide services and functionalities for the control system. Usually the life cycle of the control applications is longer than desktop applications as the machine is used for at least 10 years. Because of this long life cycle, some of the hardware components of the controllers are likely to break down and requires to be replaced. Having large spare part inventory is expensive and the size of the inventory is hard to know. Having too many parts in the inventory causes extraneous costs as the unused spare parts become obsolete when the support for the product is ended. If the inventory is too small and it runs out of spare parts during the support time, these components have to be replaced with the updated version of the component anyway. The newer versions of the components are likely to be



cheaper and have better availability. Thus, the same applications will probably be run using different kinds of hardware. The Operating System Abstraction pattern abstracts only the operating system used and thus the applications may still need recompiling or even modifications when ported from one hardware platform to other. The applications might even be impossible to port, because there is no compiler available for the programming language in the target system anymore.

**Hardware is likely to change during the long life cycle of the product and thus the application would need to be ported in order to run it with the new hardware. However, porting by recompiling the application is not always possible or desired.**

The applications in a control system typically have a long life cycle. Usually, the functionality they provide does not change between different products or product generations, but the functionality can be tuned with a set of various parameters. Thus, the actual application code does not need to be changed even if the hardware changes. For example, harvester head controller application can be used with various harvester head hardware models just by adjusting the parameters. Moreover, as technology evolves and more advanced designs become to mass-production, some of the components in the system can be replaced with new, compatible, and cheaper components. Still, the same application is used to provide the same functionality as in earlier hardware versions of the product.

Applications are executed in a specific environment. The environment consists of processor(s), memory, various input/output ports, and optional operating system to control the system. The operating system abstracts some parts of the hardware, but there still are some details, like processor's instruction set, that cannot be abstracted by the operating system. So, as the runtime environment depends on the hardware, applications need to be adapted when the hardware changes. This adaption can be almost anything from recompilation of the application to rewriting the whole application from scratch. The adaption process may be costly and this process is usually error-prone. In addition, the modifications lead to new revisions or branches of the software, which makes configuration management harder and increases costs.

To support portability of the applications, it is usually possible to provide limited backward compatibility within the same hardware product family. For example, the newer processor can run the instruction set of the older processor model in an emulation mode. However, the emulation of the old hardware device is not always implemented on any available hardware that is compatible with the work machine.

To support portability of the applications, it is usually possible to provide limited backward compatibility within the same hardware product family. For example, the newer processor can run the instruction set of the older processor model in an emulation mode. However, the emulation of the old hardware device is not always implemented on any available hardware that is compatible with the work machine.

In many cases, 3rd party hardware components are used in the system. With these components, it is usually not possible to select the desired development tools. The development tools for the 3rd party hardware may not be compatible with the control application. Still, it should be possible to use existing control applications while using the 3rd party components.

**Therefore:**

**Virtualize the runtime environment by creating a hardware independent execution platform for the application. The applications are compiled for the environment and executed in it. The runtime environment is ported for all the needed platforms.**

The virtual runtime environment (VRE) is an application, which is executed on the controller (called host system). The main idea of the virtual runtime environments is to hide the real runtime environment of the system and provide the abstracted, virtual version of the hardware to the applications (see **Fig. 3**). VRE creates a virtual representation of all the hardware required for the functionality. It is possible to have a common platform for all control applications, independent from actual hardware in use. For example, it is possible to implement a virtual CPU by having a data structure to save CPU's register values and an interpreter for the CPU's instruction set. As VRE is executed on the host system, the interpreter executes the software in the virtual runtime environment. Likewise, the bus is virtualized and isolated from the actual communication channels. In this way, the applications do not have real access to the hardware (like bus, memory or CPU), but use virtual devices instead.

VRE translates application's virtual device access to corresponding real device access. This isolation ensures that changes in the real hardware are not reflected to the application, as the application's runtime environment remains unchanged. In some cases, some of the virtual devices may not have the existing counterpart in the real system. In this case, the functionality is provided by VRE alone. For example, a virtual runtime environment could contain floating-point operations even though the CPU on the host can calculate only integer numbers. The floating-point operations are simply handled by the interpreter of the VRE. However, the emulated floating-point operations take significantly more execution time than the real ones would take.

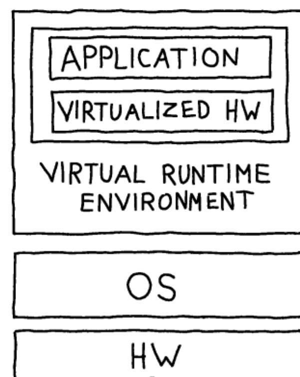



**Fig. 3.** Virtual Runtime Environment separates the application from the actual Operating System and Hardware.

The development process of the control applications for the virtual runtime environment does not differ from the process for the real runtime environment. The applications are written with the selected programming language and compiled to assembly language or bytecode for the virtual runtime environment. The properties of the runtime environment, such as the endianity of the CPU, are always known beforehand and independent from the actual target hardware when developing for VRE. Now, the developed applications need not to be recompiled as they can be executed in various host systems as long as the virtual runtime environment remains the same.

Virtual Runtime Environment is also used to support legacy platforms' hardware, which are not available anymore. In this case, the whole hardware of the platform is emulated by VRE or the suitable runtime environment is provided to execute the legacy application. For example, an embedded computer with an ARM-based CPU and real-time operating system is used to provide a virtual runtime environment for a legacy control system running a PLC application. The VRE emulates input and the output ports of the PLC and reflects their state changes to the system state variables. Now, there are two different ways to execute the actual PLC application. One is to provide emulated environment only for the application itself. In this case, VRE functions as an interpreter, which provides the same functionality for the program as the emulated PLC device by using the host computer's capabilities. The other way is to

emulate the legacy PLC device with VRE. In this latter case, all the necessary components of the PLC device have their virtual counterparts and the control logic or operating system of the PLC device is executed on the emulated hardware. In this way, the PLC program itself is executed by the emulated PLC system in the same way as it was executed with the real hardware. The latter way allows one to have all the properties of the emulated device, but the required VRE for the virtual environment is harder to implement. In addition, the former way does not allow one to have any operating system as only the application is executed by the required VRE.

There exist several commonly used commercial or open source VREs, so it may be a good idea to use them instead of writing an in-house VRE from scratch. For example, Codesys by 3S-Smart Software Solutions GmbH [CODESYS] is used in a wide range of devices. For consumer devices, Java Runtime Environment has many commercial and open source implementations for various platforms. However, if one decides to implement a virtual runtime environment, there are various publications available on this topic (e.g. (Smith & Nair, 2005) (Lain, 2006)). Virtual runtime environment is also addressed by Virtual Machine [RTDP].

Virtual Runtime Environment provides a stable environment even when the hardware may vary between different products. VRE enables one to use the same software independent from the actual platform used. This independency simplifies configuration management as the application does not need to be modified or recompiled for the new hardware. In addition, it is easier to update the hardware components, because the changes do not reflect to the application level - only VRE needs to be changed. On the other hand, VRE itself must be updated, maintained and ported to new platforms.

Legacy software can be executed even if the required legacy hardware is not available anymore. The software is executed in virtual runtime environment, which provides the functionality that the software requires.

It is possible to emulate functionality that is not provided by the hardware used. For example, floating point emulation makes processor design simpler and reduces costs. However, emulated functionalities are usually less efficient compared to the hardware-based ones.

With the virtual runtime environment, the properties of runtime environment are always known beforehand and independent from the actual target hardware. This eases the development process and makes it more robust.

The ability of emulating functionalities of existing devices can also be used to provide an environment for developing and testing where the whole system can be executed without the target device.

Because VRE emulated processor has fixed instruction execution times, the cycle rate of the application is always the same and independent of the physical processor speed. In other words, the application runs on the same pace regardless of the hardware.

Because VRE is an additional layer between the real hardware and the application and requiring additional execution time, it may be difficult or even impossible to use virtual runtime environments in the systems with very strict real time requirements. This can be compensated to some degree by having more efficient hardware, but it costs more.

When using VRE, the application is isolated from the physical devices. This eases the implementation of THIRD-PARTY SANDBOX and access control. As the application uses the virtual devices, the access rights can be checked and the operation is continued only if the application has the permission to use the device. Moreover, if DATA STATUS pattern has been applied, the status of the data can be forced to the desired value by VRE even if the application itself doesn't support data status. For example, invalid input status could always imply invalid output status and the result does not need to come from the application inside VRE.

As VRE is complex and costly to implement, it is not feasible to use it if there are only few products in the product family, the life cycle of the products is short, or the hardware is not likely to change.

An in-house development environment is used in a power plant control system. The development environment and the programming language are designed for controlling the outputs of the varistors for compensation of reactive power. The applications are executed in the virtual runtime environment as the actual execution hardware may vary. When the application is compiled, the result is a bytecode file understood by the virtual runtime environment. The runtime environment contains an interpreter that executes the code line by line. If the application changes an output value, the change is reflected to the output of the device by the virtual runtime environment.

James Smith and Ravi Nair. 2005. Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Craig, Iain D. Virtual Machines. Springer, 2006, ISBN 1-85233-969-1, 269 pages

[CODESYS] Homepage: www.codesys.com

[RTDP] B. P. Douglass: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems

## 3 Acknowledgements

I want to thank my colleagues Ville Reijonen, Marko Leppänen and Veli-Pekka Eloranta for their help. In addition, I want to thank my shepherding group in VikingPLoP 2013 and all industrial partners for their valuable cooperation in our pattern mining process: Metso Automation, Kone, Sandvik Mining and Construction, John Deere, Areva T&D, and Epec.

# 4 References

[1] Eloranta, V-P., Koskinen, J., Leppänen, M. and Reijonen, V.: A Pattern Language for Distributed Machine Control Systems, ISBN 978-952-15-2319-9, Tampere University of Technology, Department of Software Systems. Report, vol. 9, Tampere University of Techology, pp. 108, 2010.

# Patterns for Designing Programming Assignments

Samuel Lahtinen

samuel.lahtinen@tut.fi

Tampere University of Technology,
Department of Pervasive Computing
Finland

## 1 Introduction

This paper is the starting point for developing a pattern language for teaching programming techniques. Pattern mining is still an ongoing process. The set of patterns presented in this paper is aimed for teachers of software engineering courses. They can be used to aid creation of assignment descriptions. The assignments of the courses should focus on the core content of the course and encourage the students to use the techniques taught on the course. In most of the programming courses the assignments require a yearly redesign or at least yearly tuning of the assignment description. Without changes, especially the design solutions quickly become common knowledge among students and copying the implementation becomes a more tempting task as the students who have already passed the course have complete and reviewed implementations. These patterns are most suitable for advanced programming courses where the students already know the basics of programming.

The presented patterns have been mined by going through experiences on assignments of the courses on TUT Department of Software Systems. The patterns have been mined from TUT courses for programming techniques, object-oriented programming, graphical user interfaces, data structures, service-oriented systems, and artificial intelligence. We have tried to find out common properties of the successful specifications and similarly tried to find out issues related to problematic ones. Course personnel feedback and student feedback has been used as an information source. In this paper we present four patterns from a collection of patterns for designing programming assignments.

## 2 Programming assignments as a teaching tool

The purpose of an assignment is to allow the students to design and implement applications or parts of software. While doing the task the students are able to test and learn the techniques and approaches taught on the course in practice. Often a subject area of the course requires larger real-world-like applications to be sensible. If the problems are purely on a "Mickey Mouse" level, it is easier to implement the work using only the basic programming techniques. Motivating the students becomes difficult if doing the work right way, i.e. using the techniques taught in the course, is much more cumbersome.

The workload of the task is also important; the courses should follow their specified credit unit limitations. Implementing even a simple application from scratch is laborious. It is easy to create assignments where most of the effort goes to coding and testing parts of the program that have no relevance to the course itself.

The available resources are another essential issue when designing and specifying a programming assignment. Each student or group can only be given a very limited amount of personal guidance. In addition, the time the course personnel can spent on reviewing a programming assignment is limited. A course with over 200 participants requires much more exact assignment specification than a one with 15 participants. For instance, there might be a need for a common knowledge base for the most frequent problems in the assignment and templates for the grading the assignments.

In Tampere University of Technology and in universities in Finland in general lectures and weekly exercises are voluntary, only an assignment and often a final exam are compulsory. Third year and older students are often working along their studies, so even if some of the students would like to attend to teaching session, they cannot do that. On every course roughly half of the students only read assignment descriptions and other course material. Thus, the assignments are the only way to teach the students the course learning objectives and verify that they are capable of using the techniques taught on the course.

# 3 Patterns found in the pattern mining

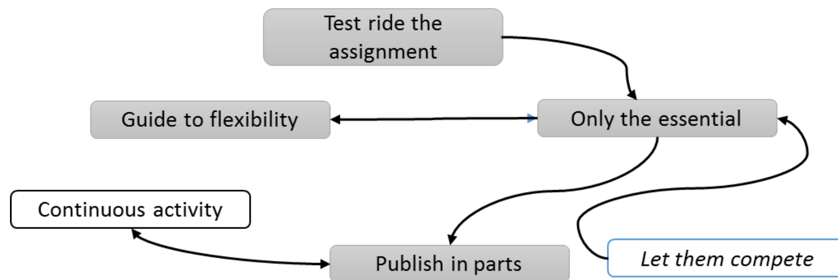



**Fig. 1.** A fragment of the pattern language

This section briefly introduces a collection of patterns that can be used when starting to design a programming assignment. First four patterns are described in detail in Section. The relations of the patterns discussed in this paper are shown in **Fig. 1**.

| Pattern name | Patlet |
|---|---|
| ONLY THE ESSENTIAL | You want to have projects that are small enough but cover the essential parts of the course.<br>Therefore: Design your assignment so that it guides the students to work on the essential core contents of the course. Provide code and components for students to ease the workload. |
| GUIDE TO FLEXIBILITY | You want your students to learn to recognize and avoid or document constrains and restrictions in their code. You want to teach to the students to write code without unnecessary constraints.<br>Therefore: Avoid exact values and numbers in your assignment description, provide initialization files and readers for them instead of giving a fixed application environment. Teach students to recognize and document their decisions. |
| TEST RIDE THE ASSIGNMENT | You have difficulties to estimate the amount of work required to complete the assignment, you have to fix and update your assignment description during the course.<br>Therefore: Use a person who has not participated on the design of the specification to implement the application. You can use her experiences and information to get |

| | |
|---|---|
| | a better estimate and more polished instructions. |
| ASSIGNMENT IN PARTS | Simple assignment works do not teach anything and demotivate advanced students, complicated assignments are too demanding for basic students.<br>Therefore: Divide the work in parts and publish them part-by-part, offer more advanced students a chance to implement extra features. |
| STUDENT SELECTED ACTIVITIES/SUBJECTS [6] | You have a fairly small group of experienced students and you do not want to limit their creativity and you do not have any brilliant ideas. Course grading is fail-pass or assignment plays only a limited role in grading.<br>Therefore: Let the students design and implement whatever they want in the scope of the course. You can review their idea and limit overambitious projects and check that projects fall into scope of the course. |
| CONTINUOUS ACTIVITY[2] | If students get an assignment and a deadline, they mostly start too late to work on the assignment. They often are not able to finish the assignment in the best possible quality and on time.<br>Therefore: include regular delivery moments of appropriate artifacts to motivate and engage the students to be active over the whole time of the assignment. These artifacts should be of value for the students |
| PERSONAL FEEDBACK | How to give the students a feeling they are doing something that interest someone? Different ways of giving feedback should be considered. See feedback patterns[8], Constant feedback [6], differentiated feedback [6].<br>Therefore: Assign the students a personal assistant, have a meeting or a feedback session early in the assignment work. The students meet their assistant and know the face behind name/email address. The students have their own assistant throughout the course for questions, grading, and final feedback. Have a final feedback session there the students can show their creation and talk about it. (Consider also peer review) |
| KEEP IT SHORT | Students hate long assignment descriptions and easily miss the key points of the assignment.<br>Therefore: Keep the core of the description short. Document the interfaces and other code you offer separately. Offer simple example codes to demonstrate the usage of components you offer. Offer a separate step-by- |

| | |
|---|---|
| | step documentation for starting the assignment. |
| LET THEM COMPETE | How to reward the best students in an open way? How to allow the students to constantly develop their solutions? You have easily measurable qualities in the assignment (e.g. performance, efficiency, A.I. competition)<br>Therefore: Set a minimum level that is required for passing, base the grading not on the result of the competition, but overall qualities. One can get the highest marks without winning being on top of the competition. Reward the best X on a list with a bonus to the grade. Update the list preferably on runtime/frequently. Make the score table visible. |
| PEER FEEDBACK/REVIEW | You have limited resources, but want to offer students as much feedback as possible. [6]<br>Therefore**:** Use peer-reviews where students or groups of students give feedback to other students. Peer-reviews cannot be used to directly grade the assignments, but they can be used to give the students additional feedback on their work. The knowledge of other students reviewing their work often encourages the students to do better work. You can also use the student feedbacks to ease your grading by verifying the main issues in the feedback are relevant. |
| GROUP WORK [6] | You have a shortage of personnel, want to have a larger assignment, or need to teach the students group working.<br>Therefore: Let the students work in a group there they can teach each other, learn group working skills, and have a larger application. |
| PAIR WORK[6] | You have limited course personnel, you want to teach e.g. benefits of revision control, or you want to have a larger assignment.<br>Therefore: Make the students work in pairs. They can help each other, they learn to co-operate, communicate, and share work. |
| SOMETHING FROM THE REAL WORLD | Using languages and environments made solely for teaching purpose or similarly languages and environments that are only used in academia tend to demotivate the students. Learning to use something no-one in industry uses is often seen useless.<br>Therefore: Include some “real-world” examples on |

| | |
|---|---|
| | your course, do not stick only to academic examples. Allow students to work with modern tools and devices. |

# 4 The patterns

In this section we present three patterns that can be employed when designing and writing a programming assignment description. This is only an incomplete subset of the patterns as this is a work in progress. The patterns use an Alexandrian(ish) pattern format [1]. The first part of the pattern is a short description of the context, the problem is given in bold. Forces are given after that and they are followed by the solution in bold. The resulting context and the consequences of the pattern application follow the solution. The applications of the pattern are given in italics as last.

## 4.1 ONLY THE ESSENTIAL

**Also-known-as:**
AVOID WASTE

You want to have assignments that are small enough but still cover the essential parts of the course.

**When working on larger programming assignments the students easily spent a significant part of their effort on work that is unrelated to the learning objectives**

**of the course. This decreases their chances to learn the essential core matter of the course.**

There is a need for a large enough programming task for the techniques of the course to be sensible. If you use overly simple applications when teaching concepts like object-oriented programming, modularity, interface classes, or implementation patterns e.g. MVC, the students do not see any benefits from applying the techniques taught on the course. They merely see the new techniques in a bad light as the same task could be implemented several times faster without using them. If the students associate the main learning objectives with a label "something laborious and useless" they need first to unlearn and when relearn to be able to master the objectives of the course.

However, if you have a larger application which the students need to implement from scratch, they are likely to spent major part of their work on coding and testing features that are irrelevant to the learning objectives of the course. The students come easily overburdened or implement their projects using only their background knowledge from previous courses and do not learn and use anything new.

Their assignment might meet the functional requirements but lack in the techniques related to the course. Thus, students get bad grades or are even unable to pass the course even though they have put large effort on the course and even submitted something seemingly good. If they are asked to revise the project to meet the requirements, it often means they need to rewrite a significant part of the application. It is hard to motivate a student to spend twice the amount of time estimated to the project. Thus some of them quit (and come again next year), others get bad grades, and many of them complain a lot (rightfully). This all means extra work for you.

**Therefore: Do not design features that are unessential to the assignment. Come up with assignment idea, pick the core learning targets, and provide implementation for the rest. Allow the students to learn to use third party code and to focus on the learning objectives instead of wasting time on implementing unessential features. Offer simple modules and interfaces to decrease the learning curve. Include a runnable sample along your code to demonstrate the usage.**

This solution can be used to increase the size of the application and thus bring the size and complexity of the project a bit closer to real software projects. For instance, on a course, where the purpose is to learn to use and implement interfaces and objects, it is sensible to provide a user interface and file handling. On the other hand, on a course on user interfaces, providing basic program control which the interface interacts makes the assignment work more meaningful. When you provide components or parts of the application, also provide a mock-up implementation which shows how to use the main features. This decreases the amount of time the students need to spend to learn to use your code. Give also all the necessary project files and guidance on how to install the tools. The students can also learn by reading code written by others.

Applying this pattern requires more time than traditional assignment where students implement everything themselves. You also need to start the preparations earlier

as you need to have the provided code files ready and tested when the assignment work begins. The time working on the provided code can be significant as you need to provide easily understandable, well-documented, and tested code. Otherwise you may end up creating extra confusion and wasted hours.

If you are not careful on choosing the provided components, you may unnecessarily limit the freedom the students have. You need to leave the students some freedom to use their creativity in design and implementation. You can also give students an option to implement provided parts by themselves. For instance, a user interface can be given on a course, but students who want to learn user interface implementation techniques can implement their own version.

You can use CONTINUOUS ACTIVITY[2] to ensure the students start to work early enough. Some of the sub-assignments can be allocated to getting familiar with the provided code. ASSIGNMENT IN PARTS is suitable for providing new parts or giving extra features as part of a setup.

Related Patterns: LARGER THAN LIFE [5], TOOL BOX[5]

*"Known applications":*

*On courses on software design and object-oriented programming the provided code includes user interfaces, utility tools, and initialization file reader. Instead of writing a file parser for some strange data related to the assignment the students can concentrate on the main application logic. Instead of having fixed values in their application, they have parts of the data given in file. Similarly, the students do not need to learn (graphical) user interface implementation details. Instead, they can concentrate design of the control logic and learn to use interfaces and modules that are provided to them.*

*On artificial intelligence course the pattern was used to provide the students game-logic a reference AI player that needed to be beaten by the students, and an interface for own AI player. The students could concentrate on the implementation of the AI without a need to implement the game itself. In addition, the game framework was used to make the students' AIs to compete against each other. The most successful ones were rewarded. (See* LET THEM COMPETE *pattern)*

## 4.2 GUIDE TO FLEXIBILITY

You have students that have passed the first programming courses where they have learned functions and basic datatypes etc. You want to help them to learn to recognize and avoid unnecessary constraints/restrictions and document the ones that are chosen on purpose. Students often design classes that just are just glorified structs and class hierarchies where inheritance is used just for type information.

**Even though you have fancy style guides and guidelines the students tend to submit code filled with undocumented restrictions and unnecessary constraints to e.g. size, length of values or objects. You need assignment descriptions that encourage the students to write more flexible code.**

Beginner coders tend to introduce undocumented restrictions and unnecessary constraints to e.g. size, length of values or objects. They tend to hardcode values and functionality on their applications. When the projects get larger these undocumented constraints are easily hidden behind interfaces and modules making the usage and modification of the components difficult.

In a more abstract level they have modules and classes which have limitations and constraints that are not documented or visible from the public interface. Giving the students feedback and even making them revise the code after their final submission is one possible approach. However, the scheduling is an issue as the course when continues to summer or long to next period. In addition, students are rarely happy to return to modify the code especially, if the modifications are not related to core subjects of the course.

Giving style guides and adding constraints and rules to the specification can be used to enforce certain features to the implementation. However, the students often tend to skip the style guides and embedding the style to the assignment description makes the specifications more annoying to read. The rules become easily shallow and are only seen as a nuisance or an extra work that has been just added to increase the workload.

**Therefore: Define the application specification so that it uses initialization files, user input etc. as source of data. Do not give everything as exact numbers, sizes, or types already in the assignment description. When applicable, do not give fixed amounts for entries or values. Teach students to recognize and document the constraints and restrictions they add to their code.**

Making the students to adapt to different settings and setups already in the design phase is a good way of guiding the students to avoid hardcoded values and unwanted restrictions. Instead of defining there can be at maximum N players, or exactly four of these, thus the loop runs till value X, they will learn to adapt to input coming from user, file etc.

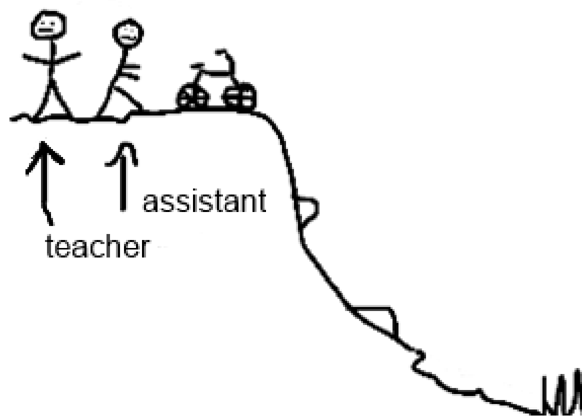
You can provide the students with tools or components to access the data. This decreases unessential work that is needed to parse the data from file or to implement UI

(ONLY THE ESSENTIAL). Consider also ASSIGNMENT IN PARTS, CONTINUOUS ACTIVITY and connect the flexibility points to new phases and extra parts published. This can underline the benefits of avoiding hard-coding and using object-oriented guidelines like SOLID[14]. Students learn the benefits when the extensions and new parts are easier to implement.

On the other hand, if the focus is solely on teaching a specific algorithm or usage of a single element in a narrow scope, need to avoid waste work is usually more significant. You should not force the students to create overly general bloated solutions for simple applications. The objective is not to teach the students solutions that are suitable for everything, good for nothing.

Related patterns:
In EXPERIENCED ADVANTAGE[12,13] pattern the problem is similar, the students do not see the advantages. The solution is to let them experience the advantages. EXPERIENCED PROBLEMS [12,13] pattern shows what happens if the given problem is not solved.

## 4.3 TEST RIDE THE ASSIGNMENT




You need an accurate estimate on the amount of work it takes to complete the project, too large assignments take time from other courses and make students work too much for their credit units and as a result students get less overall credits. Less overall credits means less money to your department... On other hand, if you make too easy assignments your students do not learn all the key aspects of the course. Thus, they can be in trouble in the following courses or when they move to working life. It also decreases the credibility of your course.

You need a clear assignment description to decrease your own work and students work when assignment deadline is closing. You want to avoid a flood of questions and requests from the students when the deadline is closing. You do not want to revise the assignment description plenty of times as you have better things to do.

**You have difficulties to estimate the amount of work required to complete an assignment and your assignment descriptions need to be revised and clarified during the course.**

Reviewing the assignment description helps to remove most of the obvious faults and inconsistencies from it. However, only when you are implementing the assignment, you'll get to the details and have to think about how all of the features. Even after the review you get loads of questions and complaints from the students and need to revise the specification and/or create FAQs and write plenty of emails and have meetings.

You may have had to make changes to the assignment requirements in the middle of the course to allow the students to complete their work. The students have trouble in understanding what to do and you waste your valuable time clarifying the description. You need to give answers that are obvious (to you) to students' questions.

As a writer of the spec you already know the unwritten details related to the work, you have already a mental image of the application. Thus, you do not need or use the assignment description. Estimating the amount of work spent on the complete project is difficult, if you try to do it by yourself. You do not have to spend time to understand the specification, to learn tools, or to learn new techniques on the course. A detailed description is more essential on the initial courses, on the advanced level you can only give rough guidelines for the course.

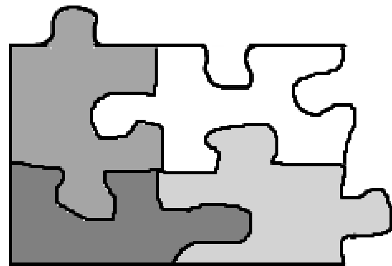
**Therefore: Use a person who has not participated on the design of the specification to implement the application- She should take notes and ask questions whenever there are is something unclear in the description or there is something essential missing.**

You need to find a person with a suitable background knowledge before you can apply the pattern. In Finland the course assistants are often students, some of them usually passed the course last year. This makes them excellent targets for the pattern. They learn a bit more about the subject, get some extra work and money, and the course will get a better assignment. In the end teacher's valuable time can be saved. Another option is to use students that ask for a possibility to pass the course outside the teaching period, e.g. in summer.

Using your own time estimate and feedback from the tester to get a better measurement for the assignment size. The feedback you can get from a person actually creating an implementation using the specification and (possibly the code files you have provided, ONLY THE ESSENTIAL) makes the assignment more polished. If you can find

errors in the provided code in time, you'll save the students' and your own time. If you plan to use automatic checking of the assignments, the testing becomes vital.

Without the testing the flaws of the assignment are often discovered only weeks before the final deadline. Making unplanned (major) changes to the assignment this late is guaranteed to invoke dissatisfaction among the students. Changing the automatic checking systems or using students to beta test it are also guaranteed ways to get peo-



ple upset.
You can also use the test implementation to develop better feedback and grading guidelines.

*Object-oriented programming courses, distributed and service oriented systems, software architecture, and basic programming have used this pattern. The usage of the solution has decreased the need to revise the assignment description, there are less errors or inaccurate workload estimates. For instance, in advanced course of object-oriented techniques features in the original assignment description (e.g. types of squares and monsters in a dungeon dwelling game) have been moved from compulsory requirements to voluntary extra bits. The student feedback on the amount of time spent on the assignment has shown the workload average has been near to the course design. Before testing there were times where the average the students spent on the assignment was 50% more than the credit unit workload.*

## 4.4 ASSINGMENT IN PARTS

You want to motivate the students to think about extensibility and modifiability. In addition you want to reward students that are interested in doing additional tasks or have put thought on design. However, giving a specification that challenges the keenest of students might be too much for your average student especially in the beginning.

**Giving a complicated and demanding assignment demotivates the normal students, and simple ones do not motivate the more advanced enthusiastic students. Large, single phase assignments do not teach the students to adapt to changes.**

You want the student to agile and to encourage them to be prepared for changes and extensions, but do not want to extend the specification in the beginning.

A specification should offer challenges and be suitable for both students that know only basics that the prerequisite courses have taught and more advanced and keen students that are interested in additional work.

The exercises and assignments should also teach about changing requirements and specifications. However, if you change the main specification, you'll annoy the students who start early and aid those who start just before the deadline.

**Therefore: Give the specification in two or more phases. Publish the extra features fairly near to the deadline to encourage students to focus on the design and to prepare for changes.**

When you publish the basic version of your assignment, state that extension will be published later containing some changes and/or additional features. This encourages the students to focus on their design and prepare for changes and extensions. If you offer some extra points for implementing the additional features and changes, the students (may) want to start their work earlier.

If the new features are published near to the deadline, implementing the complete application from scratch is almost impossible. However, those who have started early can do the extra features with a little effort.

The students who have had a sensible design are able to implement the new features easily. The rewarding experiences can be connected to the learning objectives of the course. The concrete reward in form of some extra points to the grade is only a decoy; the actual reward comes from the feeling of achievement.

If you have too specific demands you may easily restrict the development process and constrain your assignment. In addition, the style of the application needs to be doable part-by-part.

You should include a design review meeting to the assignment there you can guide the students to a right direction. This way all the students should have a fair starting point to the actual implementation phase.

You may also consider STUDENT SELECTED ACTIVITIES[6] to allow the students to pick the parts they want to implement in each phase.

CONTINUOUS ACTIVITY should be considered to be used with ASSIGNMENT IN PARTS. The delivery moments of the work can often be synchronized with publishing the new parts.

This pattern is a special case of CONTINUOUS ACTIVITY(CA). In CA the main aim is to keep the students active whole time of the assignment. In addition, smaller subtasks are not as overwhelming as one large. ASSIGNMENT IN PARTS is suitable for

courses, where software design, extensibility, modifiability belong to the key learning objectives of the course.

The pattern also shares features of STUDENT SELECTED ACTIVITIES. It allows the students to pick and implement extra bits if they want to. However, in order to pass, the students only need to do implement the main parts of the project. Similarly the majority of the grade comes from the main project and. extra parts are used to tempt and motivate the students to put more effort to early phases and the design of the project. In addition, it gives the more keen students a chance to show their skills.

The pattern shares common consequences with EXPERIENCED ADVANTAGE or EXPERIENCED PROBLEMS depending on the path they chose.

*In Object-oriented programming courses (basic and advanced) programming assignments are often games there students are given a framework. The extra features are published after the design deadline is over. Some of the features are added to provided code side, other parts should be implemented by students by specializing existing components. The approach shows the ideas behind the interface classes and traditional inheritance. Students can easily extend their applications if they have thought about the design and started implementing their application. Design feedback meetings can be used to guide the student towards designs that are missing some key features. The extra features are easy to add to games, you can easily come up with new commands, new types of squares, actions, creatures, characters, or equipment.*

## 5 Related work

Pedagogical patterns project [11] have a couple of pattern collections for teaching [6][9]. There is also an ongoing work on a pattern language for course development in computer science [5]. There are also pedagogical patterns for teaching in a foreign language [3][4] and patterns for active learning [10] a pattern collection for giving the feedback [8]. Patterns for effective teaching in seminars are also related to teaching courses [7].

This paper focuses on programming assignment design for a software engineering courses. The aim is to aid to design assignments that are suitable for both the students that self-study the course and the students who participate actively on the teaching sessions. These patterns can be taken as part of a more general course development language. The pattern collection presented in Section 3 summarizes the patterns one can consider when designing an assignment. The references to the corresponding pattern are given with the pattern. Some of the wordings have been changed to better suit the assignment design need.

## 6 Acknowledgements

I want to thank my shepherd Christian Köppe for his invaluable comments and gentle guidance to the world of patterns, my teaching colleagues for their time and discussions, and my students for acting as guinea pigs on some of the teaching experiments.

## 7 References

1. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction* (Center for Environmental Structure Series). Oxford University Press, August 1977
2. Christian Köppe. Continuous Activity - A Pedagogical Pattern for Active Learning. (July 2011), 7 pages. EuroPLoP'11
3. Christian Köppe and Marielle Nijsten. *A Pattern Language for Teaching in a Foreign Language -Part 1*. In Preprints of the 19th Pattern Languages of Programs conference, PLoP'12, 2012 16th European Conference on Pattern Languages of Programs, EuroPLoP'11
4. Christian Köppe and Marielle Nijsten. *A Pattern Language for Teaching in a Foreign Language -Part 2*. In Preprints of the 19th Pattern Languages of Programs conference, PLoP'12, 2012
5. Joseph Begin, *A Pattern Language for Course Development in Computer Science*, Pace University, http://csis.pace.edu/~bergin/patterns/coursepatternlanguage.html
6. Joseph Begin, *Some pedagogical patterns*, Pace University, http://csis.pace.edu/~bergin/patterns/fewpedpats.html
7. Astrid Fricke and Markus Voelter, *A Pedagogical Pattern Language about teaching seminars effectively*, 2000, http://www.voelter.de/data/pub/tp/html/index.html
8. Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, and Helen Sharp. *Feedback Patterns*. http://www.pedagogicalpatterns.org/
9. Joseph Bergin, *Fourteen Pedagogical patterns*, http://csis.pace.edu/~bergin/PedPat1.3.html
10. Joseph Bergin, Jutta Eckstein, Mary Lynn Manns, and Helen Sharp. *Patterns for Active Learning*. http://www.pedagogicalpatterns.org/
11. Pedagogical Patterns Project, http://www.pedagogicalpatterns.org/
12. Christian Köppe, *A Pattern Language for Teaching Design Patterns (part1)*, 16th European Conference on Pattern Languages of Programs, EuroPLoP'11, 2011
13. Christian Köppe, *A Pattern Language for Teaching Design Patterns (part2)*, Preprints of the 18th Conference on Pattern Languages of Programs, PLoP'11, October, 2011
14. Robert C. Martin, *Designing Object Oriented Applications using UML, 2nd. ed., Prentice Hall, 1999*

# Patterns for Messaging in Distributed Machine Control Systems

Marko Leppänen

{firstname, lastname} @tut.fi
Department of Pervasive Computing
Tampere University of Technology Finland

## 1 Introduction

In this paper we will present two patterns for sharing information in distributed machine control systems. A distributed machine control system is a software entity that is specifically designed to control a certain hardware system. This special hardware is a part of a work machine, which can be a forest harvester, a drilling machine, elevator system etc. or some process automation system. Some of the key attributes of such software systems are their close relation to the hardware, strict real-time requirements, functional safety, fault tolerance, high availability and long life cycle.

Distribution plays a major part in the control systems. Different functional hardware parts of the machine are physically apart from each other and their corresponding control software is usually located in a embedded controller node near the controlled hardware. The nodes must communicate with each other in order to perform their functionalities. It is also common that the system nodes have very wide variety in their computational capabilities. Usually the system has several simple embedded controllers with limited computational abilities also known as low-end nodes. In addition to these embedded controllers the system may contain one high-end node that has processing power that is comparable to a common desktop PC. Due to these facts, a distributed control system needs to distribute information between different parts of the system. The information-sharing and messaging capabilities of such systems is discussed in these patterns in more detail.

The patterns in this paper were collected during years 2008-2011 in collaboration with industrial partners. Real products by these companies were inspected during architectural evaluations and whenever a pattern idea was recognized, the initial pattern drafts were written down. These draft patterns were then reviewed by industrial experts, who had design experience from such systems. After these additional insights, and iterative repetitions of the previous phases, the current patterns were written down. We hope that the final pattern language can be tested on implementation of some real system after all patterns in the language are published.

The published patterns are a part of a larger body of literature, which is not yet publicly available. All these patterns together form a pattern language, which consists

Marko Leppänen p. 1, 2013.
© 2013

of more than 80 patterns at the moment. A part of the pattern language in this paper is presented in a pattern graph (see Fig 1.) to give reader an idea of how these selected patterns fit in the language. These two patterns are closely related in the pattern language and therefore are ideal to be submitted together as a whole. In the following sections, all the pattern names are written in SMALL CAPS.

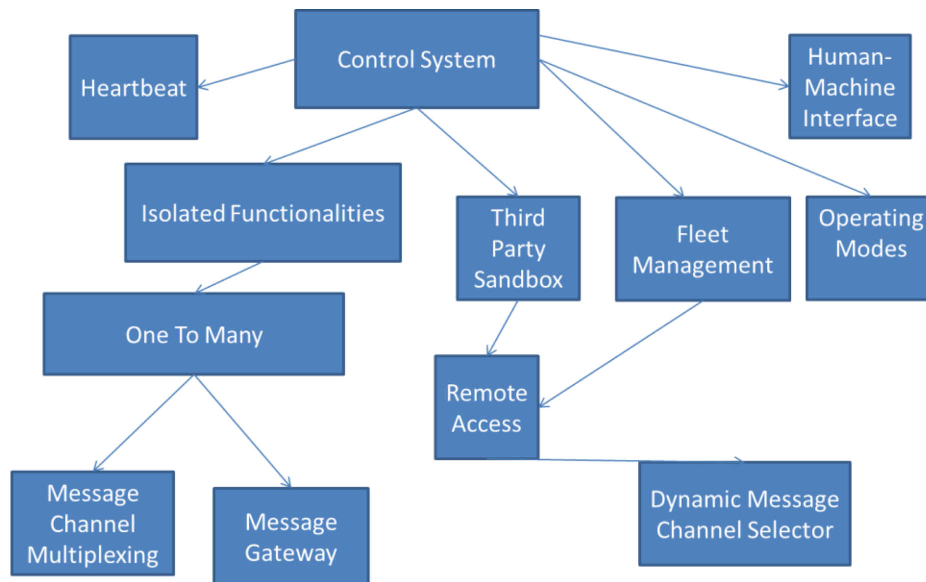



**Fig. 1.** The relations between patterns mentioned here

In the second section, we will first introduce our pattern language and the pattern format. Following this, the selected two patterns are presented in detail. Finally, the last sections contain the acknowledgments and references.

## 2 Patterns

In this section, a set of two patterns is presented. Together, these patterns are a part of a sub graph in the pattern language in Fig. 1 The pattern graph is read so, that a pattern is presented as a box in the graph and an arrow presents a connection between the patterns. The connection means that the pattern from which the arrow emerges is refined by the pattern that the arrow points to. In other words, if the designed system still has some unresolved problems even after some pattern is applied, the designer can look to the refining patterns for yet another solution if they want solve the current design issues. The patterns refine each other extending the original design with other solutions.

For example, the CONTROL SYSTEM pattern is the root of the whole pattern language and it is referenced in the following patterns. So, the CONTROL SYSTEM is the central pattern in designing distributed control systems. It presents the first design problem the system architect will face: Is a control system needed in this context? Table 1 presents all patterns that are shown in Fig. 1 and all the patterns that are referenced later on in this paper.

| Pattern name | Description |
|---|---|
| CONTROL SYSTEM | Implement control system software that controls the machine and has interfaces to communicate with other machines and systems. |
| ISOLATED FUNCTIONALITIES | Identify logically connected functionalities and compose these functionalities as manageable sized entities. Implement each of these entities as their own subsystem. |
| ONE TO MANY | Build a network called a bus where all nodes share the same communication medium. Nodes send information as messages over this medium. All nodes can receive all messages from the network and will see if there is currently anything relevant on the bus. |
| MESSAGE GATEWAY | Add a component, a message channel gateway, to the system between message channels. This component routes message traffic between message channels. If needed, the component can filter messages according to specific criteria defined in the system configuration. In addition, the component handles the translation from a message protocol to another. |
| MESSAGE CHANNEL MULTIPLEXING | Separate communication channel from the actual physical bus by creating virtual channels. Virtual channels might be multiplexed in one physical channel using dividing the channel into time slots or can be divided over several physical buses. |
| FLEET MANAGEMENT | Implement a Fleet Management application and install it on-board the machine. Within that application, create common interfaces and information model for all work machines to manage them as a fleet. Production information, which conforms to the information model, can be transferred to and from the machine using the common interface. In this way, the machine can coordinate the optimization of work with other machines via an ERP system. |
| REMOTE ACCESS | Add a remote connection gateway on-board which enables communication between the machine and the remote party. The remote connection gateway transforms the used messaging scheme to suit the local and remote parties' needs and can take care of authentication. |
| THIRD PARTY SANDBOX | Provide an interface and tools for third-party application developers. Third-party applications can use the machine services only through this interface so that they will not interfere |

| | |
|---|---|
| | with the machine's own applications. The interface provides common ways to access data and services. |
| DYNAMIC MESSAGE CHANNEL SELECTOR | Organize all communication channels using the wanted properties, e.g. cost of communication. Add a component which automatically changes the communication channel if the higher priority channel is not available. |
| HUMAN-MACHINE INTERFACE | Add a human-machine interface. It consists of ways of presenting information and controls to manipulate the machine. These typically are displays with GUIs, buzzers, joysticks and buttons etc. The way of presenting the information and the controls of the machine must be decoupled, e.g. with a Message Bus. |
| OPERATING MODES | Design system so, that it consists of multiple functional modes. These modes correspond to certain operating contexts. The mode only allows usage of those operations that are sensible for its operating context. |
| HEARTBEAT | Make a node to send messages at predetermined and regular intervals to another node. The other node knows the message interval and waits for the message. If the message does not arrive in time, the remedying actions can be started. |

**Table 1.** Patlets

Our pattern format closely follows the widely-known Alexandrian format [1]. First we present the context for the problem. Then, the problem is concentrated in a couple of sentences that are printed with a bold font face. After that, a short discussion about all forces that are affecting the problem is given. In a way, it is a list of things to consider when solving this problem. Then, after "Therefore:" the quick summarization of the solution is given. Then, after a three star transition line, the solution is discussed in a detail. This section should answer all the forces that were left open in the previous section. Then another star transition marks the end of the section. This section describes briefly the consequences of applying this pattern. After the last star transition a real life example of the usage of this pattern is given.

## 2.1 One to Many

...there is a CONTROL SYSTEM with ISOLATED FUNCTIONALITIES, and thus the system is divided into several nodes. As the nodes have to collaborate, every node has to be connected to all those nodes from which it needs information. Similarly, the node has connections to all other nodes which use the information it provides. Usually these connections are dedicated wires where communication is carried out. Basically

two-way communication requires at least one wire, creating a mesh (see Figure 2) where the communication requirements of each node form a connection to other nodes. The nodes are coupled to each other by this extensive wiring. If the system design evolves during the system lifetime so that the communication requirements change, it would propagate changes to the wiring, too. Alternatively, the nodes should have routing capabilities in their software which would allow the sender to reach the recipient using intermediating nodes.´

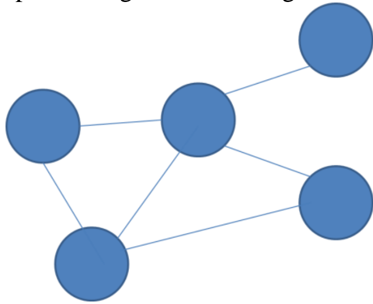
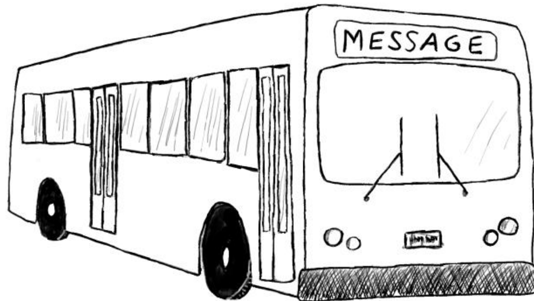


**Fig. 2.** A communication mesh with extensive wiring between nodes

**Every node has to know how to reach the recipients of information it produces and this forms a tight coupling between nodes. If the communication requirements change, redesign of wiring or software on several nodes is needed.**




Physical wiring for communication nails down the communication structure. However, the work machine design may change over time, as new hardware with different capabilities can be added to the original design and the software may evolve to have more optimized algorithms. Thus, the communication needs of the devices may change as information may be produced and consumed in unforeseen places in the system. In such evolving design fixing the wiring is too rigid solution. Adding wires afterwards is laborious and sometimes even impossible as the cable raceways might be already full. In addition, it may be impossible or difficult to add new devices to the system after the design phase as the wiring harness only has connectors in predefined places.

In some cases, the same CONTROL SYSTEM application should be executable on products that have slightly different hardware, but still belong to same product family. In order to allow software reuse, the communication infrastructure should be flexible enough.

The communication infrastructure should be scalable, so that new participants may join the information exchange. The communication should allow both increase in amount of communicating parties and amount of information sent by a participant in the communication. However, amount of wiring to be used for information transfer should be minimized. The wiring is relatively expensive, adds weight and takes up space. The more wire there is, more difficult it is to design the wiring so that it won't be prone to electromagnetic interference or breaking. In addition, the assembly of the machine should be fast on the assembly line. Extensive wiring is slow and error-prone to install as assembly line personnel need to install multiple wires. So the wiring for the communication between nodes should consist only of minimal number of wires to allow easy installation.

There should be a uniform way to communicate with different nodes on the system, so that the developer does not need to be interested in the details of communication when designing the applications. Furthermore, it is crucial that the communication way does not depend on with which other node is recipient party to avoid errors in the development phase.

**Therefore:**

**Build a network called a bus where all nodes share the same communication medium. Nodes send information as messages over this medium. All nodes can receive all messages from the network and will see if there is currently anything relevant on the bus.**

Communication between nodes should be carried over a shared medium, to where all nodes are connected to. This medium usually consists of a single cable, which is connected in a bus network topology. In rarer cases, the communication can be electromagnetic waves sent over air. In these cases, sender node transmits the message over the radio waves and all other nodes may receive all sent messages, as radio waves reach all receivers.

Typically, however, a bus topology is used and the communication medium connects the communicating devices to each other and allows sending information to the recipients, see Figure 3. The connecting cabling creates the physical layer that is the foundation for the data exchange. On this layer the information is presented by voltage/current changes that are interpreted as binary data. The data forms messages which all nodes should be able to understand. The physical layer and messages are

usually implemented using a commercial solution, such as CAN bus [2], as many problems that arise when designing the communication on physical level have already been addressed. There exist other physical topologies, such as a star configuration. A star topology may be used, if all branches of the star get all the messages, but, for example, in the case of CAN this requires the system to include an active component, e.g. a hub or a switch in the middle of the star.

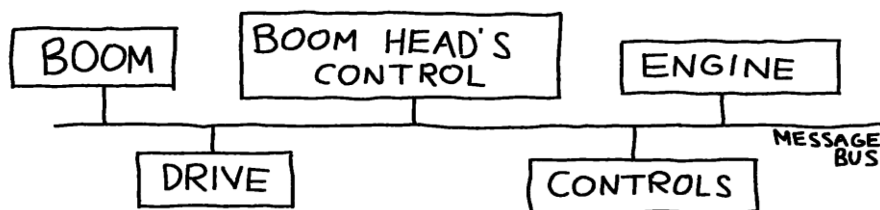



**Fig. 3.** X: An example of bus topology network

Now, every node can listen to the communication medium and pick up relevant messages. The sender does not have to know which other nodes are interested in its messages, only which information it is supposed to publish. This information is encapsulated in messages which are broadcast to all nodes. For example, as in Figure 3, an operator uses controls, such as a joystick, which sends control messages to the message bus. The boom controller can read the broadcast message and move the boom accordingly. The recipient usually does not need to know the actual sender. It just reads every message from the bus and if the message does not contain information the node needs, it just simply discards the message. This abstracts the physical location of the devices from the control applications. As the only interface between the nodes is the messages, it is quite easy to move functionality from one node to other, or even nodes from one location to another.

When designing the connection with wires, same things should be taken into consideration as when installing any other cabling. For example, one should consider external forces that may break the wiring. However, the situation is usually remedied by the fact that a communication cabling consists of only few wires and machine joints can be passed through using sliding ring connections.

If two or more nodes try to communicate at the same time a collision occurs and thus the actual bit stream on the bus becomes garbled. When the messaging is carried out in a point-to-point fashion, the collision may happen only when both parties of the channel send their messages simultaneously. In shared communication channels, collisions may happen frequently as any message sent on the bus reserves the communication channel for a certain time span. The probability of a collision increases when the amount of nodes on the bus grows. However, in a shared channel, the probability can be diminished if the nodes listen to the bus simultaneously when sending the mes-

sages. If there is message traffic on the bus, the node must retain from sending its own messages. However, if one node starts transmitting messages too often, the other nodes will have difficulty to get their messages through as they must wait for a silent period. The problem is commonly called the babbling idiot's problem [3] and can be addressed by using Bus Guardian pattern [4].

Waiting for the silent moment does not alone solve the collision problem completely. Several nodes may start sending their message still exactly at the same moment when they have detected that the message channel is silent. These messages collide and the information does not get through. In some bus technologies, the collision causes both senders to wait for a random time and try again. If yet another collision occurs as the nodes waited for the same time by chance or a third-party is willing to send its messages, the collided nodes double their maximum waiting period. This mechanism is called binary exponential back-off [5]. It suits badly real-time applications as an unfortunate sender may have a long and nondeterministic wait time.

The collision problem can be remedied by comparing any sent data bit to the actual bit on the bus. If the bit on the bus is high when the node is sending a low bit, the node should immediately stop sending as the discrepancy means that there is another node sending another message at the same time. Now the message has not yet been garbled and the other node will get its message through. The CAN bus utilizes this mechanism so, that in the beginning of the message there is the sender id, which at same time defines the priority of the node. High priority node's identification will get through as its bit pattern has earlier dominant bit. As sharing the communication channel is the root cause of the collisions, the only way to be sure that the collisions will not happen is to remove the sharing by using MESSAGE CHANNEL MULTIPLEXING.

One can gain the benefits of mass-production by selecting a bus standard that is widely used in the industry. These standard solutions typically have solved the problems related to messaging in such way, that it is suitable for a certain domain. Some examples in the machine control domain include CAN bus, FLEXRAY [6], Local Interconnect Network [7], and PROFIBUS [8] using multi-drop EIA-485 as the physical connection standard. As there are several vendors adhering to the same standards, there are ready-made devices that support the standard communication, for example, sensors that can be attached. It makes the system designers' jobs easier as they can use ready-made devices and software instead of proprietary solutions. However, using commercial solutions may cause unwanted dependencies.

Usually selecting one commercial communication protocol also affects the hardware and vice versa. Vendor lock-in may easily occur when all components should be acquired from one vendor. The long life cycle may further amplify these problems as the support for a certain communication solution may end over years. Proprietary components may have limited availability in the area where the work machine is used. Thus, it may be difficult to acquire spare parts for commercial off-the-shelf solutions

in some parts of the world. In some cases, the commercial standards have many stakeholders and the development of the communication solution may be led by companies from other domains. Thus, the development is driven by other industrial requirements than your own and these requirements might weigh some other quality attributes of the system design more than what would be optimal for your design. For example, CAN bus is very heavily-driven by automotive industry and its applications have different requirements than in the work machine industry as cars need quick response time, but the amount of information is smaller. However, in a work machine there might be more data to be transferred, but slower response time would be adequate.

This pattern is an example of PUBLISHER/SUBSCRIBER pattern [9]. It is also documented in the context of DDS middleware as ONE TO MANY [RTI]. MESSAGE CHANNEL [8] and MESSAGE BUS [9] describe similar mechanisms for building communication channel between nodes but in different domains.

Nodes on a bus may communicate with each other and the communication infrastructure will be scalable in the amount of nodes and flexible in the locations where information is produced and consumed. For example, if the physical location of a sensor or an actuator changes in the design, it is easy to accommodate the changed messaging requirements for these nodes. In addition, the recipient is not usually interested in actual location of the other node. Location transparency allows additional flexibility in connecting the hardware devices to the connectors. However, the physical properties of the bus and the amount of collisions in the messaging may set a maximum for the amount of the nodes on one bus segment.

As every node listens to all messages on the communication medium, they can also act as a monitoring point inspecting the condition of the message channel. Thus, more of nodes there are on the bus, the smaller chance of residual errors remain on the communication, making the system safer.

It is easier to design the wiring of the system as all the communication is carried out over just a few wires. The production costs are cut down as fewer wires are needed to be installed on the machine and every device does not require its own set of wires. However, the message bus acts as a single point of failure, but as the communication protocol may provide a HEARTBEAT service, communication failures are usually easy to detect. For safety reasons, it is usually better not to try to communicate at all when the reliability of the message channel is compromised.

As the nodes do not subscribe per se to any messages, but it is rather their responsibility to read interesting messages from the bus, the sender cannot know if the message has been delivered to any interested parties, if the message channel is not reliable. Thus, some mechanism to acknowledge the delivery is usually needed. For ex-

ample, in the case of CAN bus, the other nodes acknowledge all correct messages they were able to receive by overwriting sent recessive ACK bit as dominant. Still, the sender will not know if anyone really used the message and more elaborate acknowledgement mechanisms may be needed.

The selected communication protocol and the physical bus may set constraints to the communication infrastructure. For example, the transfer rate and the length of the bus may be limited. In some cases, the maximum length of dropdown lines and the minimum distance between the nodes in order to avoid signal reflections from the ends may become an obstacle of physical cabling design. Even the wiring consists of only a few communication wires, which are easy to extend to reach all over the machine, physical connectors are still needed near the location of the new node or device.

In some cases, the actual physical layout of the system makes it hard to use bus topology on the system. For example, if the system consists of two clearly separated locations where nodes reside, it may be difficult to connect these isolated groups of nodes with a cable. Then it would be sensible to segment the bus into two separate parts and connect them with a MESSAGE GATEWAY.

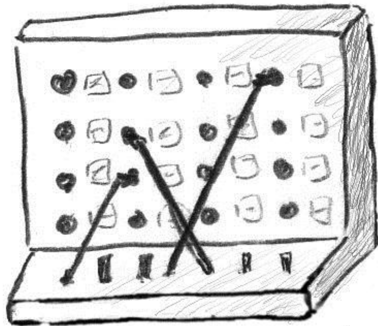
A truck has multiple controllers as the system design dictates that the controllers of the subsystems should be located as close as possible to the actual hardware of the subsystem. This is done to minimize the amount of wiring needed for actuators and sensors. For example, the engine controller is mounted in the engine compartment on the engine itself. The controllers should work in co-operation and thus the units need to communicate with each other. The nodes are connected with a CAN cable which consists of three signal wires, i.e. CAN HI, CAN LOW and ground wires, terminated from both ends. This cabling allows the nodes to send SAE J1939 (SAE International, formerly Society of Automotive Engineers) messages to other nodes [10]. SAE J1939 protocol is designed so, that all messages are broadcast to other nodes. This makes it easy to accommodate additional nodes to the bus. For example, if a trailer is attached to the truck, the nodes on the truck and trailer may communicate with each other. If the sender needs to specify the receiver, the protocol allows adding the destination address to the message. The messages have also a 29-bit PGN (Parameter Group Number) field that tells the receiver the purpose of this message and allows the recipient to quickly determine what kind of data this message contains. The PGN field discriminates if the message is for a specific recipient or a broadcast message. In addition, it contains the source address of this message and in a specific part of the header the rate of transmission and the message priority. This special part of the header also includes the data assignment of the parameter part of the message. The parameter part has the actual payload of the message. A certain parameter group has same data length in bytes, data type, resolution, offset, range and a reference label or tag. The SAE J1939 also allows multipart messages and defining new parameters. Sending

longer messages than the maximum length of a frame (8 bytes) is possible by using higher level services. It also has address claiming mechanism and diagnostics built-in.

### 2.2 Dynamic Message Channel Selector

...there is a distributed CONTROL SYSTEM which has a REMOTE ACCESS allowing accessing the machine resources remotely. There are multiple technologies, e.g. wireless LAN, satellite telephone or GPRS (General Packet Radio Service) connection, supported which offer ways to communicate with the machine remotely. As the mobile work machine can be situated on various work sites all around the globe, various environmental factors may interfere with the communication channels. For example, if a forest harvester works in a stand situated far in the wilderness, there are no terrestrial base transceiver stations nearby. Thus, if a remote connection is needed, only expensive satellite phone connection is possible. On the other hand, when the harvester is at the factory perimeter, there might be Wireless LAN available for high bandwidth data transfer. In another case, a mining drill is situated in an underground mine, where WLAN availability varies depending on the location as the massive rock walls block the signal. The location of the machine and available communication channels affect the band-width, communication costs, transfer rate and other attributes and constraints of the communication channel. This makes the communication scheme design hard as there are multiple trade-offs depending on the environment.

**There can be several communication channels that could be used. However, under certain circumstances some of these channels are not available. Still, the most cost-efficient communication channel available should be chosen.**



There are plenty of remote communication technologies available and it is quite cheap to implement several of them on a machine. The implementation can be done with commercial off-the-shelf hardware either as an additional chip on a controller or as a separate device that can be plugged directly to the message bus. It is reasonable to support multiple technologies as the wireless technologies have different kinds of

communication properties and restrictions. For example, some provide communication that needs dense base station infrastructure, but has high transfer rate and so on.

The information that is stored on the machine has varying importance and urgency for the remote party. Having a high importance means that the remote party definitely needs to have access to the information. Conversely, low importance means that the remote party can manage without this single piece of information and it would only be needed for optimization purposes. On the other hand, high urgency information means that it is needed on the remote end within a short time interval from the moment when it is produced or otherwise it will be obsolete. Low urgency means that the information will be still relevant after a long time span. Thus, the urgency factor is essentially a time-to-live value for the information. For all information, the communication quality should be optimized in terms of urgency and importance.

All the information must be conveyed as messages that are sent via the communication channel. As the information consists of varying amounts data, some messages can be larger than some other messages. However, the message size is not correlating to the importance or urgency. As the machine moves, the set of available messaging options changes dynamically. There can be situations where there is no communication channel available for some period of time, and in some other cases, it might be that a certain communication channel is never available for a single machine.

**Therefore:**

**Organize all communication channels using the wanted properties, e.g. cost of communication. Add a component which automatically changes the communication channel if the higher priority channel is not available.**

Organize the communication channels according to the properties you wish to optimize in order to achieve the best possible cost-efficiency. Consider all different communication properties, i.e. security of the channel and the possibility of eavesdropping, cost per sent amount of data, bandwidth, reliability, stability, latency and so on. Organize this information in a form of an array; see Fig 4 for an example. Some of the properties can be dynamic as in the case of availability. The unit of the properties can be, for example, an integer value from one to ten describing your view of the property for this channel. For example, in the Fig 4, Iridium has low latencies compared to WLAN, but it has really high price per sent unit. In addition to these channel properties, one should take into account the nature of the data that has to be sent. Every nugget of data may have differing urgency, importance, security etc. requirements. The weighing of certain properties may also change depending on the operating context, such as the OPERATING MODES of the machine. Now, define a utility function that sets weights for different communication properties when given the wanted properties for the data and the operating context.

| | AVAILABLE | COST | STABILITY | SECURITY | LATENCY | BANDWIDTH | RELIABILITY |
|---|---|---|---|---|---|---|---|
| LAN | 0 | 0 | 10 | 10 | 10 | 20 | 9 |
| WLAN | 0 | 1 | 5 | 5 | 7 | 10 | 3 |
| 3G | 1 | 4 | 7 | 4 | 4 | 4 | 5 |
| IRIDIUM | 1 | 15 | 10 | 7 | 1 | 1 | 7 |

$f_{UTILITY}$ (URGENCY, IMPORTANCE , PROPERTIES[ ])

**Fig. 4.** An example with four different wireless technologies with different properties. The utility function uses these properties as one parameter.

Now, the designed utility function will then return the most cost-efficient channel for communicating the data. Of course, the utility function must take in consideration the availability of channels as there might be no connectivity because of the environmental factors, as the terrain, location and such. See Figure 5 for an example system, where wireless channels are organized so that a certain data nugget has high urgency, importance, and reliability requirement will be sent using Iridium satellite connection. The other option, 3G link, would be too slow and unreliable for this data transfer.

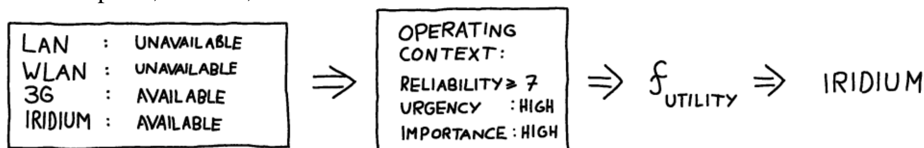



**Fig. 5.** an example of a utility function, which takes reliability, urgency and importance requirements for a certain data nugget as parameters.

After the utility function has been devised, add a component to the REMOTE ACCESS service which uses the utility function to select the best channel available according to the criteria. The component will send the messages through the best available option. If there is no sensible option for sending the data, the messages which have great importance but low urgency may be stored locally on the machine so that they could be sent when the cost of transfer will be feasible again. High urgency data must be discarded, if the sending delay would grow too large.

One way to organize data transfer is to stop communication altogether for certain services. For example, if a 3rd party software needs updating, it might be reasonable to block this happening if the communication channel is slow and/or expensive. See Figure 6 for an example how different services could be grouped. When the communication channels' availability changes, the component selects the next suitable option from the list of available communication channels and makes the proper changes to the amount of messages which can be sent through this channel.

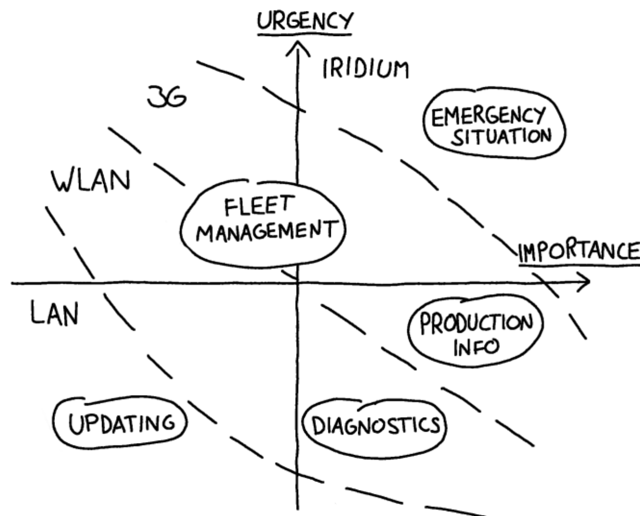



**Fig. 6.** an example how to weigh different services according to their importance and urgency.

If the data transfer does not cost by amount of transferred data, the selector can send non-urgent messages via channels by splitting the data into smaller chunks which are sent in the background. This takes more time, but the transfer does not disrupt the available bandwidth. The DYNAMIC MESSAGE CHANNEL SELECTOR may be used even if there are two similar channels for use in order to maximize the bandwidth. For example, if two radio channels are available to connect the machine to the remote party, they both can be used so long as the usage does not cost anything extra. This maximizes the amount of data that can be sent. If the other channel is not available any more, the data transfer is not disrupted, it just continues with lower bandwidth.

In some cases, DYNAMIC MESSAGE CHANNEL SELECTOR can be used in local communication too. Usually this is the case, if there is for example, an Ethernet cable and CAN bus connecting two nodes. Ethernet is used in transferring huge amounts of data, e.g. diagnostics, and CAN is used for control. Now, if the CAN bus is severed, Ethernet could dynamically be used to carry out some communication. However, this approach is limited to Limp Home kind of functionality only as there is no determinism in the communication anymore.

The messages over the remote link are delivered using the optimal channel. This may save communication costs or provides the most reliable channel for the data. In

special cases, the machine might switch to more secure channel in order to prevent eavesdropping.

If several channels can be used in parallel, the availability of messaging and its bandwidth may be optimized. In some cases, the messages can be sent in chunks, so that momentary unavailability of a channel won't disturb the communication as a whole.

A rock crusher sends its production data to the FLEET MANAGEMENT. The production information consists of rock type, its volume and diagnostics information about the velocity of the transfer conveyer belt, jaw speed and so on. On the crusher's HUMAN-MACHINE INTERFACE, there is a configurable setting, which allows the user to decide properties for the production information that is sent over the wireless link. The wireless link can be established as a WLAN or 3G link, depending on the situation. The operator selects that the rock type and volume has high importance, but low urgency. The selection is done as the production information is essential for the later stages of the processing chain, but it is possible to send the day's production information as a batch in the end of the work shift. On the other hand, the jaw and belt information has low importance, but high urgency as the run-of-the-mill data is needed only in calculating preventive maintenance needs for the crusher. However, if the belt or the jaw jams, the situation has to be quickly notified to the maintenance team.

As the crusher usually is located in an open-cast mine, it seldom has a WLAN connection which would have enough bandwidth to send all the data. If the machine currently has not a WLAN access, it will buffer the production data to be sent later on. Only critical messages, such as if the machine becomes incapacitated, are sent over the expensive 3G link.

## 3 Acknowledgements

I especially wish to thank my colleagues Dr. Johannes Koskinen, Veli-Pekka Eloranta and Ville Reijonen for their valuable help and input during the gathering process of these patterns. I also wish to thank all industrial partners for their willingness to provide the opportunities for the pattern mining. These companies include Areva T\&D, John Deere Forestry, Kone, Metso Automation, Sandvik Mining and Construction, Creanex, Rocla, SKS Control and Tana. In addition, I would like to thank Nokia Foundation and Pirkanmaan rahasto for their scholarships and grants which have aided us in writing these patterns.

## Bibliography

[1] C. Alexander, The Timeless Way of Building, Oxford University Press, 1979.

[2] ISO, "Road vehicles -- Controller area network (CAN)," ISO.

[3] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Norwell, MA: Kluwer Academic Publishers, 1997.

[4] W. Herzner, W. Kubinger and M. Gruber, "Triple-T (Time-Triggered-Transmission) - A System of Patterns for Reliable Communication in Hard Real-Time Systems," in *Proceedings of EuroPLoP 2004*, 2004.

[5] IEEE, "IEEE Standard 802.3-2008," IEEE, 2008.

[6] FlexRay Consortium, "Flexray communications system - protocol specication version 2.1 revision a," 2005.

[7] ISO, "LIN SPECIFICATION 2.2.A., ISO 17987 Part 1-7".

[8] IEC, "IEC 61158/61784".

[9] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley, 1996.

[10] SAE International, "SAE: J1939 Standard," SAE International, Warrendale, USA.

[11] F. Buschmann, K. Henney and . D. C. Schmidt , Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, Wiley, 2007.

# Catalog of Safety Tactics in the light of the IEC 61508 Safety Lifecycle

Christopher Preschern, Nermin Kajtazovic, and Christian Kreiner

Institute for Technical Informatics, Graz University of Technology, Graz, Austria
christopher.preschern@tugraz.at, nermin.kajtazovic@tugraz.at,
christian.kreiner@tugraz.at

**Abstract.** Safety tactics describe general architectural design decisions and their effect on the overall system safety. Currently these safety tactics do not directly address the consequences of design decisions on safety certification.
To establish this connection, we refine safety tactics by extracting information concerning architectural design decisions from the IEC 61508 safety standard. We generalize this information in order to describe the effect of safety tactic usage on different development phases of safety-critical systems. We provide the whole revised catalog of safety tactics and we show its application by analyzing the Triple Modular Redundancy design pattern regarding its safety tactic usage to evaluate the effect of the pattern on safety certification.

**Keywords:** safety tactics, IEC 61508

## 1 Introduction

Safety standards contain information about requirements which have to be fulfilled to achieve functional safety certification. Often some methods and architectures for fulfilling the safety requirements are suggested in the standard and in practice just these, sometimes outdated, methods and architectures are used. The introduction of new methods and architectures requires proof of their validity regarding functional safety which can be a tedious task and can increase certification costs significantly. There is no general evaluation of methods and architectures which allows to evaluate them regarding safety certification in order to aid the certification of novel concepts.

Safety patterns address this problem in a way that they describe the consequences of applying a specific architecture; however, they cover a rather specific and implementation focused view of this problem. To cope with the problem on a more general level, we evaluate the consequences of safety-related architectural design decisions (safety tactics) on safety certification. We examine existing safety tactics and discuss their suitability for the IEC 61508 safety standard. We mine architectures and methods suggested in the IEC 61508 standard regarding the tactics they use and regarding their effect on different phases of the safety lifecycle. Based on our analysis of used tactics in the IEC 61508 standard, we

re-organize and re-structure existing safety tactics to be more intuitive. Furthermore, we refine the safety tactics by describing their influence on different safety lifecycle phases in general and more specific by relating IEC 61508 methods to the tactics. We present the refined catalog of safety tactics and we apply it to an example where we analyze the consequences of the Triple Modular Redundancy (TMR) pattern [1] on safety certification.

This paper is organized as follows. Section 2 gives an introduction to the IEC 61508 safety lifecycle and focuses on its realization phase which is later on analyzed for the tactics. Section 3 introduces the idea of tactics and Section 4 gives an overview of current tactics in the safety domain. Furthermore, in this Section we discuss why and how existing safety tactics should be modified. In Section 5 we present the tactic catalog with focus on the tactic influence on safety certification. Section 6 analyzes the TMR safety pattern by using the refined safety tactics. Section 7 gives an extended overview of related work on architectural tactics with focus on safety tactics. Section 8 concludes this work and gives an outlook on the future potential of this work.

## 2 IEC 61508 Safety Lifecycle

The safety lifecycle according to IEC 61508 provides a process framework which allows to achieve functional safety for a product by following the methods and requirements posed by the standard for each phase of the lifecycle. An overview of the lifecycle is shown in Figure 1.

The planning phases addressing the overall product safety include definition of concept and scope, a hazard and risk analysis resulting in safety requirements, and the allocation of Safety Integrity Levels (SILs) to components. During the planning phases, plans for the operation, maintenance, installation, and safety validation have to be defined. An important phase of the safety lifecycle is the product realization phase, which distinguishes between hardware and software implementation and can be divided into the following sub-phases:

- Requirements specification - Full specification of safety-related functions for the product, allocation of SILs to these functions, and specification of risk reduction measures for these functions.
- Validation planning - Preparation of a plan how to validate the system against the specified safety requirements.
- Design and development - Design and implementation of the safety-critical software/hardware according to the safety requirements.
- Integration - Integration/Assembly of developed software/hardware subsystems to form the complete safety-related product.
- Operation and maintenance - Activities to ensure the proper operation of the developed software/hardware product (does not cover system modifications).

In the phases after the realization of the system, the plans on operation, maintenance, installation, and safety validation have to be carried out. Additionally,

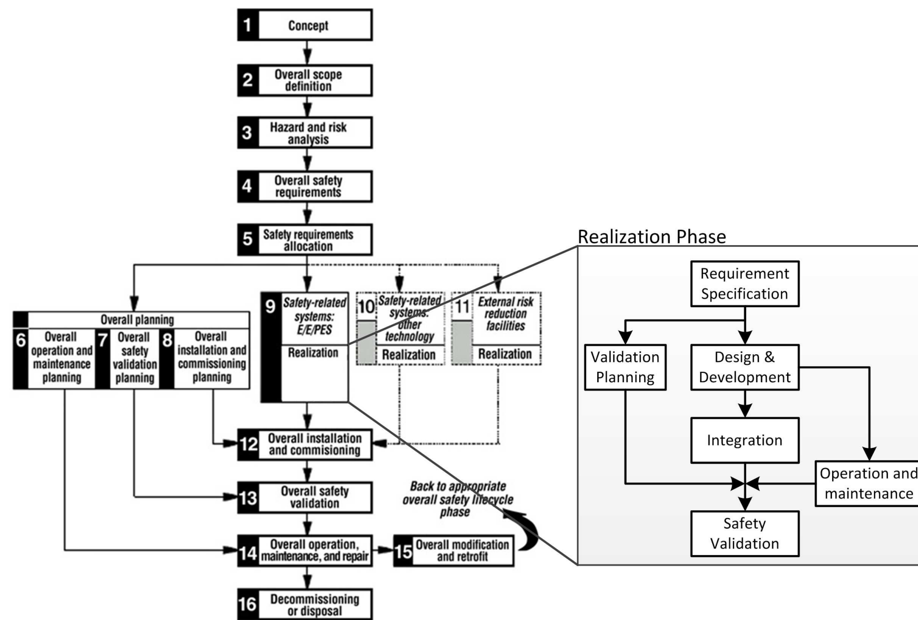



**Fig. 1.** IEC 61508 safety lifecycle (incl. realization phase) [2]

other phases of the lifecycle address product modifications, decommissioning, and disposal.

In this paper we focus on the effects of architectural safety tactics on the product realization phase and the following phases like safety validation and product modification. We do not describe the effect of architectural safety tactics on all of the phases mentioned above, because when analyzing the safety standard, we did not find relationships of the tactics to all of the safety lifecycle phases, especially not to the early phases.

## 3 Introducing Tactics

Tactics are architectural design decisions which influence and manipulate quality attributes [3]. Compared to design patterns, they describe general concepts or principles and do not describe solutions for a problem in a given context. For example, the Voting tactic describes how to achieve failure containment by choosing an appropriate system output from redundant system components. Compared to patterns, the tactic is more general and does not describe a specific solution but rather provides the underlying idea for possible solutions. In this case, a possible solution could be the TMR pattern which uses the Voting tactic to choose for the majority of three redundant subsystem outputs. Usually, a tactic can be found in several architectures or patterns and can even be seen as building blocks for design patterns [4].

It is difficult to keep tactics and patterns apart as there is no clear boarder between the two. However, Ryoo et al. [5] specify some criteria to identify tactics. For a design decision on order to be a tactic, it has to be atomic. This means that it cannot be divided into other multiple tactics, however it can be refined. For example, the Redundancy tactic is refined by the Replication Redundancy tactic and the Diverse Redundancy tactic, but it is not composed of them. Furthermore, Ryoo et al. say that tactics focus on a single quality attribute (e.g. safety) and patterns usually affect several quality attributes.

## 4 Safety Tactic Catalog

A collection of safety tactics presented by Wu [6] is shown in Figure 2. These tactics were mined from safety architectures described in literature and address failure avoidance, failure detection, and failure containment.

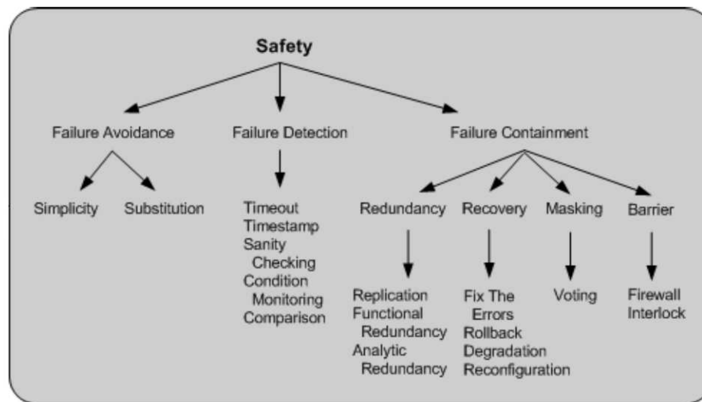



**Fig. 2.** Safety tactics proposed by Wu [6] (arrows show tactic refinements)

We analyzed methods and architectures used in the IEC 61508 standard and related them to Wu's safety tactics. Part 7 of the IEC 61508 standard explains several safety-related methods and architectures and describes their aims. We linked them to Wu's safety tactics by manually searching for similarities between the method or architecture aims and the tactic aims.

For some tactics we could not find any relationship to the standard at all and some methods and architectures had very similar relationships to the same set of tactics indicating that these tactics are rather similar. Furthermore, some of the tactics describe rather specific safety-related solutions (e.g. Timestamp), while others describe general concepts (e.g. Voting). This motivated us to revisit the safety tactics, to make the safety tactic catalog more intuitive. We add tactics we found rather often in the IEC 61508 standard to the catalog and we skip

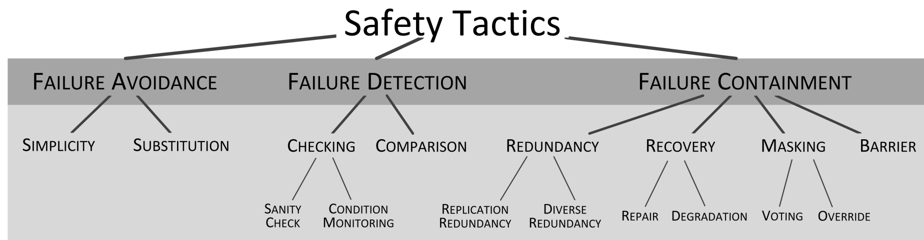



**Fig. 3.** Re-organized safety tactics (arrows show tactic refinements)

tactics which we did not find in the standard or which were very implementation-specific.

The detailed description of each tactic we use in the following section and the detailed process how we manually analyzed the IEC 61508 standard will be given later on in Section 5.

### 4.1 Re-organized Safety Tactics

Figure 3 shows our re-organized safety tactics catalog. We keep Wu's general categorization of safety tactics in failure avoidance, failure detection, and failure containment tactics. We do not modify the failure avoidance tactics, because methods regarding failure avoidance in the IEC 61508 standard could perfectly well be mapped to Wu's tactics. However, we change parts of the failure detection and failure containment tactics as explained in the following.

Wu's safety tactics Sanity Check and Condition Monitoring check a system state or value against additionally introduced redundant information. The difference is, that Sanity Check introduces this information in the specification, while Condition Monitoring introduces the information in the implementation phase. Due to the similarities of the two tactics, we generalize them in a Checking tactic. A similar tactic was already suggested in [7] where Wu's safety tactics were also slightly adapted.

We recognized that just very few IEC 61508 methods used the Timestamp or Timeout tactic. This lead to the idea that they might be rather specific tactics and not very general. The Timeout tactic detects excessive time-resource usage. This is a simple check of the time condition compared to a specified limit and can be considered as a Sanity Check. The Timestamp tactic checks the validity of an entity by checking a timestamp attached to it, which also is a Sanity Check of a beforehand specified time condition. We therefore see the Timeout and Timestamp tactics as refinements of Sanity Check; however, we do not include them in our tactic collection because we want to focus on more general tactics. We are not the first to eliminate Timestamp and Timeout from the safety tactics collection; also in [7] these tactics are omitted.

Wu distinguishes between three types of redundancy: replication (redundant identical hardware), functional (redundant implementation), and analytical (re-

dundant specification). The methods from the safety standard which we linked to functional and analytical redundancy are very similar. Therefore, we combine these two types of redundancy and call it Diverse Redundancy.

No methods of the IEC 61508 standard were mapped to Wu's Rollback or Reconfiguration tactics, because they rather address availability concerns which are covered by availability tactics [8]. However, several parts of the standard suggest recovery from errors by Repair and Degradation which we include in our tactic catalog.

We added the Override tactic to Masking, because the safety standard often describes fail-safe mechanisms, which differ from the Voting tactic. These mechanisms are based on output decisions of redundant channels where a specific output state (safe state) is preferred to other states.

The Interlock and Firewall tactic are very implementation-specific and similar mechanisms are described in literature as patterns (Output Interlock pattern [9], Firewall pattern [10]). Therefore, we omit these tactics.

## 5 Refined Tactics Catalog

In this section we present the catalog of safety tactics and discuss their consequences on different phases of the safety lifecycle. We structure each tactic into the sections **Aim**, **Description**, **Influence on the Safety-Lifecycle**, and **Related IEC 61508 Methods**. We refine Wu's safety tactics mostly with information from the seven parts of the IEC 61508 standard [2]. We studied the standard to find links between parts of the standard and the safety tactics. We started with part 7 of the standard which contains a collection of techniques which are often applied in the safety domain. We mapped these techniques to the safety tactics by finding similarities between the technique aims and the tactic aims. The techniques serve as the main source for the **Related IEC 61508 Methods** section of the tactics. We also generalized information about the description and the aim of the techniques to refine the **Description** and **Aim** sections of Wu's tactics.

The techniques of part 7 are often referenced in other parts of the standard, especially often in parts 2 and 3. We analyzed the context of these references to find out further information about the tactics and their effects on different parts of the safety lifecycle. From the safety lifecycle described in Section 2, we just present the phases directly influenced by the safety tactics, which are: Specification, Design and Development, Integration, Operation and Maintenance, Modification, Verification, and Safety Validation. The effect of tactics on these parts of the safety lifecycle is given in the **Influence on the Safety-Lifecycle** section of the tactics and is mainly based on the parts 2 and 3 of the safety standard.

Additionally to the above mentioned approach to mine the IEC 61508 standard for safety tactics, we also went through the parts 1-6 of the standard again from the beginning to the end to find any connections to the safety tactics. This yielded a very similar result to the above mentioned approach. It only differed

in a few additional connections between the standard and the tactics mostly coming from part 6.

Now we present the refined safety tactic catalog.

## 5.1 Failure Avoidance

Simplicity and Substitution are failure avoidance tactics. If applicable, they are often preferred to failure detection and failure containment tactics [11], because they are rather independent from other tactics and do not create overhead for other safety lifecycle phases.

Simplicity

**Aim** - Avoid failures through keeping the system as simple as possible.

**Description** - Simplicity reduces the system complexity. It includes structuring methods or cutting unnecessary functionality and organizes system elements or reduces them to their core safety functionality, thus, eliminating hazards. An example for the application of the Simplicity tactic is an emergency stop switch system which is usually kept as simple as possible.

**Influence on the Safety-Lifecycle** - The tactic reduces effort for every phase in the safety lifecycle due to reduced system complexity or even reduced system functionality. However, most other safety tactics contradict Simplicity, because they require additional system components (e.g. a voter) which are not absolutely necessary for the core system functionality. In particular for early phases Simplicity enables significant complexity reduction. When applied during the specification phase, it increases understandability and predictability of the system behavior (IEC 61508-3 Annex F). For the Design&Development phase, it enables easier system development which is required in IEC 61508-3 7.4.2.2, 7.4.3.6, 7.4.2.6 and 7.6.2.2. However, the tactic might also put constraints on system development. For example, IEC 61508-3 7.4.4.13 requires to limit the programming language command set to the usage of safe, well-proven commands.

**Related IEC 61508 Methods** - IEC 61508-7: B.2.1 structured specification, B.3.2 structured design, C.2.7 structured programming, E.3 structured description method, C.4.2 programming language subset, C.4.2 limit asynchronous constructs, E.5.13 software complexity controller

Substitution

**Aim** - Avoid failures though usage of more reliable components.

**Description** - Components or methods are replaced by other components or methods one has higher confidence in. For hardware and software this can mean usage of existing components which are well-proven in the safety domain.

**Influence on the Safety-Lifecycle** - Changing software or hardware components can require re-doing the safety hazard analysis [6]. However, software

components can also be exchanged with previously developed components or third-party components to reduce the certification effort by re-using certification knowledge or documents for these components. SUBSTITUTION can increase hardware or third-party component costs if safer components are used. For example, buying a SIL3 component usually is more expensive than buying a SIL2 component.

**Related IEC 61508 Methods** - IEC 61508-7: B.3.3 usage of well-proven components, B.5.4 field experience, C.2.10 usage of well-proven/verified software elements, E.20 application of validated soft-cores, E.35 application of validated hard-cores, E.41 usage of well-tried circuits, C.4.3 certified tools and compilers, C.4.4 well-proven tools and compilers, E.4 well-proven tools, E.42 well-proven production process, E.28 application of well-proven synthesis tools, E.29 application of well-proven libraries

### 5.2 Failure Detection

Every failure detection method requires some kind of redundancy and testing of the redundant information. The CHECKING tactics introduce diverse information to check a system and the COMPARING tactic compares fully redundant information or systems.

CHECKING - SANITY CHECK

**Aim** - Detection of implausible system outputs or states.

**Description** - The SANITY CHECK tactic checks whether a system state or value remains within a valid range which can be defined in the system specification or which is based on knowledge about the internal structure or nature of the system. An example for a SANITY CHECK is a stuck-at fault RAM-test which checks the proper functionality of the memory during system runtime. The test is based on the understanding of the memory behavior (if we write data to the memory, we should later on be able to read the same data). Faults are detected if the memory behaves differently.

**Influence on the Safety-Lifecycle** - Plausible system outputs and states have to be specified (e.g. IEC 61508-3 C.2 3a where preconditions limit the system input range). This value range limitation can help during the system verification, because just the defined value range has to be tested (IEC 61508-3 C.2). For safety validation it can be argued that the SANITY CHECK introduces a diverse implementation for checking the safety functionality and therefore detects random as well as systematic implementation or design faults to some extent (IEC 61508-6 D.1.4).

**Related IEC 61508 Methods** - IEC 61508-7: A.1.2 monitoring relay contacts, A.2.7 analog signal monitoring, A.3.1-A.3.3 self-tests, A.4.1-A.4.4 checksums, A.5.1-A.5.5 RAM-Tests, A.6.1 test pattern, A.7.1 one-bit hardware redundancy, A.7.2 multi-bit hardware redundancy, A.7.4 inspection using test patterns, A.9 temporal and logical program monitoring, C.3.3 assertion programming, C.5.3 interface checking, C.4.1 strong typed programming language

Checking - Condition Monitoring

**Aim** - Detect deviations from the intended system outputs or states.

**Description** - Condition Monitoring checks whether a system value remains within a reasonable range compared to a more reliable, but usually less accurate, reference value. The reference value is computed at runtime by a redundant part in the implementation which can be based on system input values and is not pre-known from the specification (like it would be the case for Sanity Check). An example for Condition Monitoring is a system which has to be time-synchronized via the Internet and which checks if the synchronized time is feasible by comparing it to an internal clock.

**Influence on the Safety-Lifecycle** - An additional element providing the reference value has to be implemented. In general, the Condition Monitoring tactic implies more development overhead than Sanity Check. Condition Monitoring primarily protects from random faults. However, if it uses a diverse implementation for monitoring the safety functionality, also systematic implementation or design faults can be detected (IEC 61508-6 D.1.4).

**Related IEC 61508 Methods** - IEC 61508-7: A.1.1 failure detection by online monitoring, A.6.4 monitored outputs, A.8.2 voltage control, A.9 temporal and logical program monitoring, A.12.1 reference sensor, A.13.1 monitor

Comparison

**Aim** - Detection of discrepancies of redundant system outputs.

**Description** - Comparison tests if the outputs of fully redundant subsystems are equal in order to detect failures. The Comparison tactic usually implies the usage of a redundancy tactic. An example for the application of the Comparison tactic is a dual-core processor running in lock-step mode. The processor runs the same software on both cores and compares their outputs after each cycle.

**Influence on the Safety-Lifecycle** - An additional element which requires resources at runtime to compare the subsystems has to be implemented.

**Related IEC 61508 Methods** - IEC 61508-7: A.1.3 comparator, A.6.5 input comparison/voting

## 5.3 Failure Containment

Failure containment describes ways how to handle failures which are recognized by failure detection. The Masking and Barrier tactics prevent failures from affecting other parts of the system and the Recovery tactics deals with correcting failures. The Redundancy tactics provide multiple systems which are necessary for some other tactics.

Redundancy - Diverse Redundancy

**Aim** - Introduction of a redundant system which allows detection or masking of failures in the specification or implementation as well as random hardware failures.

**Description** - Diverse Redundancy can be applied to the specification or to the implementation level. In a system using Diverse Redundancy on the implementation level, redundant components use different implementations which were developed independently from the same specification. Diverse Redundancy on a specification level goes one step further and additionally requires that even the requirement specifications for the redundant components have to be set up by individual teams.

**Influence on the Safety-Lifecycle** - Diverse Redundancy highly contradicts the Simplicity tactic, because the additionally introduced redundant systems require a lot of effort (multiple effort for specification, implementation, verification, modification, ...) which does not add to the system functionality. If redundant systems are used, then it has to be shown for safety validation that the systems are independent from each other which can be achieved by application of the Barrier tactic (IEC 61508-1 7.6.2.7). Redundant hardware systems can more easily be validated for safety, because for a system with no hardware fault tolerance, diagnostic tests have to be run each time before computing a safety-critical function. This requirement is not so strict for hardware redundant systems (IEC 61508-2 7.4.4.1.4, 7.4.4.1.5, 7.4.4.2.1, 7.4.5.3).

**Related IEC 61508 Methods** - IEC 61508-7: A.7.6 information redundancy, A.13.2 cross-monitoring of multiple actuators, B.1.4 diverse hardware, C.4.4 diverse programming

Redundancy - Replication Redundancy

**Aim** - Introduction of a redundant systems which allows detection or masking of random hardware failures (not systematic failures).

**Description** - Replication Redundancy means introduction of a redundant system of the same implementation. The redundant systems maintain the same functionality, use identical hardware, and run the same software implementation. An example for Replication Redundancy is the RAID1 data storage technology.

**Influence on the Safety-Lifecycle** - Replication Redundancy requires multiple effort for hardware installation and modification. If redundant systems are used, then it has to be shown for safety validation that the systems are independent from each other which can be achieved by application of the Barrier tactic (IEC 61508-1 7.6.2.7). Redundant hardware systems can more easily be validated for safety, because for a system with no hardware fault tolerance, diagnostic tests have to be run each time before computing a safety-critical function. This requirement is not so strict for hardware redundant systems (IEC 61508-2 7.4.4.1.4, 7.4.4.1.5, 7.4.4.2.1, 7.4.5.3).

**Related IEC 61508 Methods** - IEC 61508-7: A.2.1 tests by redundant hardware, A.2.5 monitored redundancy, A.3.5 reciprocal comparison by software, A.4.5 block replication, A.6.3 multi-channel output, A.7.3 complete hardware redundancy, A.7.5 transmission redundancy

Recovery - Repair

**Aim** - Bring a failed system back to a state of full functionality.

**Description** - The full system functionality is manually or automatically restored if a system failure occurs.

**Influence on the Safety-Lifecycle** - A Repair or Degradation tactic is necessary for all non-redundant hardware elements which maintain a safety functionality (IEC 61508-2 7.4.8.2). However, complex recovering systems like self-reconfiguring systems are not recommended by the standard (IEC 61508-3 A.2) and make validation more complicated.

**Related IEC 61508 Methods** - IEC 61508-7: C.3.9 error correction, C.3.10 dynamic reconfiguration

Recovery - Degradation

**Aim** - Degradation brings a system with an error into a state with reduced functionality in which the system still maintains the core safety functions.

**Description** - Degradation systems define a core safety functionality. The systems maintain this safety functionality and additional non-critical functions. In case of an error, the system falls back into a degraded mode in which it just maintains the core safety functionality. An example where the Degradation tactic is often applied are automation systems. These systems control safety-critical processes and often visualize these processes in a GUI. If the system has too few resources (e.g. processing time), the system stops the GUI service and just focuses on its core functionality to control the safety-critical processes.

**Influence on the Safety-Lifecycle** - Degradation mechanisms for the system have to be specified (IEC 61508-2 7.2.3.2) and a Repair or Degradation tactic is necessary for all non-redundant hardware elements which maintain a safety functionality (IEC 61508-2 7.4.8.2). Degradation can decrease the safety validation effort, because just the degradation mechanism and the core safety functionality have to be validated. Additionally, the tactic fulfills the requirement of the standard to describe a well defined behavior in case of errors (IEC 61508-2 7.2.3.2, IEC 61508-3 7.2.3.2).

**Related IEC 61508 Methods** - IEC 61508-7: A.8 voltage supply error handling, C.3.8 degraded functions

Masking - Voting

**Aim** - Mask the failure of a subsystem so that the failure does not propagate to other systems.

**Description** - Voting makes a failure transparent. The tactic does not try to repair the failure, but it hides the failure through choosing a correct result from redundant subsystems. It decides for the majority of the output values.

**Influence on the Safety-Lifecycle** - In order to apply Voting, a redundancy tactic has to be used and a voter element has to be implemented. Subsystems of a voting system can be repaired while in operation, because the overall system can still operate if a subsystem is under repair (IEC 61508-6 B.3.1). However, voting systems are not as safe as systems which just

compare their results and ensure a safe state if any of the results differs (IEC 61508-6 B.3).

**Related IEC 61508 Methods** - IEC 61508-7: A.1.4 voter, A.6.5 input comparison/voting

MASKING - OVERRIDE

**Aim** - Mask the failure of a subsystem so that the failure does not propagate to other systems.

**Description** - The OVERRIDE tactic forces the system output to a safe state. For example, if we have a system which is in a safe state when shut off, we can apply the OVERRIDE tactic to shut off the system if we have doubt about the system output (e.g. if an output validity check fails). In this scenario overriding the system output with a safe output value decreases the availability of the system. Another form of the OVERRIDE tactic, which does not decrease the availability and is closely related to the VOTING tactic, chooses the output of redundant subsystems by preferring one subsystem or one output state over another.

**Influence on the Safety-Lifecycle** - A preferred system output state has to be defined and an override mechanism has to be implemented. OVERRIDE systems are easier to validate, because they follow the fail-safe principle (IEC 61508-1 7.10.2.6).

**Related IEC 61508 Methods** - IEC 61508-7: A.1.3 comparator, A.1.5 idle current principle, A.6.5 input comparison/voting, A.8.1 overvoltage protection with safety shut-off, A.8.3 power-down with safety shut-off

BARRIER

**Aim** - Protect a subsystem from influences or influencing other subsystems.

**Description** - The BARRIER tactic provides a mechanism to protect from unintentional influences between subsystems. To apply BARRIER, the interfaces between subsystems have to be analyzed and specified. These interfaces are controlled at runtime by a trustworthy component (the BARRIER) which often is an already existing reliable mechanism. An example for a BARRIER is a memory protection unit which controls and restricts the communication between different tasks.

**Influence on the Safety-Lifecycle** - The interfaces between subsystems have to be specified. According to IEC 61508-3 8.3.1, non-safety related functions should be separated from safety-related functions, which can be achieved by the BARRIER tactic. It can also aid the SIMPLICITY tactic by structuring the system (IEC 61508-2 7.2.2.1). BARRIER enables modular safety certification and modification and can reduce the validation effort if it is proven that the subsystems cannot unintentionally influence each other which has to be shown by an effect analysis (IEC 61508-3 C.8, Annex F).

**Related IEC 61508 Methods** - IEC 61508-7: A.11 separation of energy lines from information lines, B.1.3 separation of safety functions from non-safety functions, B.3.4 modularization, C.2.8 information hiding/ encapsulation, C.2.9 modular approach, E.12 modularization, C.3.11 time-triggered architecture

| | | | Specification | Design & Development | Operation & Maintenance | Verification & Validation | Modification |
|---|---|---|---|---|---|---|---|
| Failure Avoidance | Simplicity | | reduced effort due to simpler system | reduced effort due to simpler system, can imply functionality restrictions | - | easier to validate | - |
| Failure Avoidance | Substitution | | - | re-use of well-proven, reliable components, higher costs for reliable third party components | - | easier validation due to lower ecpected failure rate | - |
| Failure Detection | Checking | Sanity Check | requires redundant information from the specification | - | runtime comparison requires additional resources | - | - |
| Failure Detection | Checking | Condition Monitoring | - | requires redundant parts of the implementation | additional resources for runtime comparison and diagnostic system | - | - |
| Failure Detection | Comparison | | - | - | runtime comparison requires additional resources | safer than Voting | - |
| Failure Containment | Redundancy | Diverse Redundancy | multiple system specification effort | multiple system implementation effort | multiple resources, multiple installation and maintenance effort | requires Barrier, multiple V&V effort | multiple effort for HW and SW |
| Failure Containment | Redundancy | Replication Redundancy | - | - | multiple resources, multiple installation and maintenance effort | requires Barrier, multiple V&V effort | multiple effort for HW redundancy |
| Failure Containment | Recovery | Repair | - | - | Self-repairing systems need additional resources | Self-repairing systems are difficult to validate | - |
| Failure Containment | Recovery | Degradation | define degraded state | - | - | well defined core behavior in case of errors → easier safety validation | - |
| Failure Containment | Masking | Voting | - | - | systems can be maintained while in operation | less safe than simple failure detection | systems can be modified while in operation |
| Failure Containment | Masking | Override | requires definition of a preferred safe state | - | - | | - |
| Failure Containment | Barrier | | supports separation of safety/non-safety functions | - | subsystems can be maintained independently | provides subsystem independence, easier V&V through modular certification possible | enables modification of subsystems without re-verification of the whole system |

**Fig. 4.** Safety tactics and their effect on different phases of the safety lifecycle

### 5.4 Overview and Discussion of the Safety Tactics

Figure 4 gives an overview of our re-organized safety tactics and their influence on the safety lifecycle and presents relationships between the tactics. The information is mostly based on the *Influence on the Safety-Lifecycle* parts of the tactics described in the previous section.

The revised version of the safety tactics provides more consistency compared to Wu's tactics. The problem with Wu's Timestamp and Timeout tactic as special case of Sanity Check is resolved.

Just few methods and architectures from the IEC 61508 standard address failure containment tactics. We think that the reason why just few failure containment tactics were found in the safety standard is that some of the tactics, such as Masking for example, are more concerned with availability than with safety. Therefore the standard does not focus on these tactics.

## 6 Refining the TMR Pattern by Reasoning with Tactics

In this section we use our refined safety tactics to discuss the safety-related effects of applying the TMR architectural pattern.

The TMR architecture shown in Figure 5 uses three channels and compares the outputs of the channels. A voter decides for the output value which is given by at least two of the channels. The architecture therefore allows one channel to be erroneous while still maintaining full system functionality. In our example we assume simple hardware replication with identical software running on the channels.

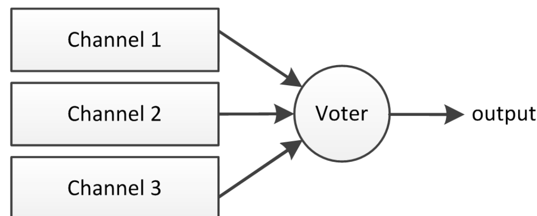



**Fig. 5.** Homogeneous TMR architecture

The TMR architecture described above uses two general safety tactics: Redundancy and Masking. More specific, Replication Redundancy is used, because there are identical redundant channels and Voting is used to mask errors of a single channel. If we have a look at Figure 4, we can see that the Replication Redundancy tactic requires the Barrier tactic during the Verification&Validation phase of the safety lifecycle. This means that to design a TMR system in the safety domain, also the Barrier tactic has to be considered right at the beginning of the architecture design in order to assure that the three subsystems do not influence each other in terms of common cause failures. This information might be obvious to a safety domain expert, however, for unexperienced system architects such information can be crucial.

| | Specification | D&D | Operation & Maintenance | Verification & Validation | Modification |
|---|---|---|---|---|---|
| Replication | - | - | multiple resources, multiple installation and maintenance effort for HW | requires Interlock, multiple V&V effort | multiple effort for HW redundancy |
| Voting | - | - | systems can be maintained while in operation | less safe than simple failure detection | systems can be modified while in operation |
| Barrier | supports separation of safety/non-safety functions | - | subsystems can be maintained independently | provides subsystem independence, easier V&V through modular certification possible | enables modification of subsystems without re-verification of the whole system |

Table I: Safety tactics for the homogeneous TMR architecture

We end up with three tactics which are used by the TMR system: Replication Redundancy, Voting, and Barrier. Table I shows the TMR relevant tactics taken from Figure 4. We can see that our TMR architecture influences the Operation&Maintenance phase in a way that multiple hardware is required and

has to be installed. This implies multiple hardware maintenance effort. However, the three hardware channels can be maintained independently and they can even be maintained during operation due to the BARRIER and VOTING tactics. For safety validation, random hardware channel failures are independent and systematic failures are not detected. The VOTING tactic requires validation of the correct functionality in case of an error and is therefore more difficult to validate than simple systems which shut down or go to a safe state in case of errors. Just like with maintenance, hardware modifications for single channels can be done during operation. Multiple effort is required if the modification affects the redundant channels. The effort for system specification and development is not increased due to the simple usage of replication.

If we look at the detailed tactic descriptions from Section 5, we can get further information for the TMR pattern in terms of a quick reference to the IEC 61508 standard. As one example, the REPLICATION REDUNDANCY tactic is connected to IEC 61508-2 7.4.4.1.4 which says that self-tests for a single channel do not have to be executed each time before the execution of a safety function if redundant channels are present. It is sufficient to execute the self-tests once a day. Such quick references provide us with very detailed IEC 61508 related information.

The evaluation of the TMR pattern through the usage of our refined set of safety tactics leads to much more detailed information regarding safety, in particular safety certification, than existing safety pattern catalogs such as [12] offer.

## 7 Related Work on Safety Tactics

In this section we present related work on architectural tactics with focus on safety tactics. We also present patterns which are related to the IEC 61508 standard.

Bachmann et al. introduce the idea of architectural tactics and describe their relation to system quality attributes [3]. They present a collection of tactics for availability, security, testability, usability, modifiability, and performance. Wu and Kelly extended this collection by adding a set of tactics for the safety quality attribute [6] [13]. They further develop an approach how to apply safety tactics by stating anti-requirements which can be handled by the application of safety tactics [14]. This approach is explained in more detail in [15], where a whole architectural safety-reasoning framework is presented.

Another approach of how to reason about the usage of safety tactics is presented in [16] and [11], where safety attributes of a system are evaluated by risk-based qualitative reasoning. This reasoning is done before and after the application of a safety tactic in order to evaluate the applicability of the tactic. The application of safety tactics in order to build a safe architecture is also described in [17] and [7] with focus on the integration of the tactics into the V-model which is commonly used for IEC 61508 system development.

To the best of our knowledge there is no work directly relating safety tactics to a safety standard so far, however, Armoush [12] constructs an extensive catalog of safety patterns and evaluates them regarding IEC 61508 safety certification by presenting the applicability of a pattern according to recommendations in the safety standard. Compared to our work he does not discuss the influence on the safety system over the whole safety lifecycle and does not give much detail regarding the influence on safety certification. [18] covers organizational patterns for IEC 61508 software development. They focus on patterns for software development and not on the relation of IEC 61508 to architectural patterns.

## 8 Conclusion

In this paper we provide a revised catalog of safety tactics and relate these tactics to the IEC 61508 safety standard. This allows us to evaluate generic architectures like safety patterns regarding their effect on safety certification during different phases of the safety lifecycle. With the connection between safety tactics and the IEC 61508 standard it is now easier to provide a system architect with information about the safety related consequences of choosing a specific tactic or pattern. Here, an advantage of the safety tactics is that compared to the safety standard, they provide system architects with a view of the safety domain, which is more familiar to them. The tactic catalog therefore provides a good source of information for early architectural decisions for systems which have to be safety certified.

The re-organized set of safety tactics can serve as a basis for future work on refining patterns in the safety domain. Future work could also include refining our tactics or evaluating them with respect to a different safety standard. We believe that our re-organized version of safety tactics builds a mature set of safety tactics and that system developers can use them to argue for the safety of their system during safety certification.

## ACKNOWLEDGMENTS

We would like to thank our shepherd Jari Rauhamäki who significantly helped to improve this paper. He provided us with good overall feedback and in particular helped to improve the tactic descriptions of this paper with his detailed safety-related knowledge.

## References

1. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Pearson (2002)
2. International Electrotechnical Commission: IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety Related Systems (2010)

3. Bachmann, F., Bass, L., Klein, M.: Deriving Architectural Tactics : A Step Toward Methodical Architectural Design. Technical Report March, Carnegie Mellon Software Engineering Institute (2003)
4. Kumar, K., Prabhakar, T.V.: Pattern-oriented Knowledge Model for Architecture Design. In: 17th Conference on Pattern Languages of Programs (PLoP). (2010)
5. Ryoo, J., Laplante, P., Kazman, R.: A Methodology for Mining Security Tactics from Security Patterns. In: 2010 43rd Hawaii International Conference on System Sciences, IEEE (2010) 1–5
6. Wu, W.: Safety Tactics for Software Architecture Design. Master's thesis, The University of York (2003)
7. Hill, A., Nicholson, M.: Safety tactics for reconfigurable process control devices. In: 4th IET International Conference on Systems Safety 2009. Incorporating the SaRS Annual Conference, IET (2009)
8. Bass, L., Clements, P., Rick: Software Architecture in Practice. 2 edn. Addison-Wesley (2003)
9. Rauhamäki, J., Kuikka, S.: Patterns for control system safety. In: Proceedings of the 18th European Conference on Pattern Languages of Programms (EuroPLoP '2013). (2013)
10. Schumacher, M.: Firewall Patterns. In: Proceedings of the 8th European Conference on Pattern Languages of Programms (EuroPLoP '2003), Universitaetsverlag Konstanz (2003)
11. Im, T.: A Reasoning Framework for Dependability in Software Architectures. PhD thesis, Clemson University (2010)
12. Armoush, A.: Design patterns for safety-critical embedded systems. PhD thesis, RWTH Aachen University (2010)
13. Wu, W., Kelly, T.: Safety tactics for software architecture design. In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC), IEEE (2004) 368–375
14. W. Wu and T. Kelly: Managing Architectural Design Decisions for Safety- Critical Software Systems. In: Quality of Software Architectures, Springer (2006) 59–77
15. Wu, W.: Architectural Reasoning for Safety- Critical Software Applications. PhD thesis, University of York (2007)
16. Im, T., Vullam, S., McGregor, J.: Reasoning about Safety during Software Architecture Design. In: Proceedings of the 19th International Conference on Software Engineering and Data Engineering (SEDE 2010). (2010)
17. Hill, A.: Safety Tactics for Reconfigurable Process Control Devices. Master's thesis, University of York (2008)
18. Vuori, M., Virtanen, H., Koskinen, J., Katara, M.: Safety Process Patterns in the Context of IEC 61508-3. Technical report, Tampere University of Technology (2011)

# Patterns for safety and control system cooperation

Jari Rauhamäki, Timo Vepsäläinen and Seppo Kuikka

Tampere University of Technology, Department of Automation Science and Engineering
P.O. Box 692, FI-33101 Tampere, Finland,
{jari.rauhamaki|timo.vepsalainen|seppo.kuikka}@tut.fi

## 1 Introduction

In this paper three patterns for safety and control system cooperation are presented. Here a safety system refers to a functional safety system. Functional safety systems are systems dedicated to retain the safety of humans, environment and property, e.g. the machine itself. Such a system may, for instance, implement a stopping of a work robot when human enters the working area of the robot. Safety system implements safety functions that lower risk related to a certain event to a tolerable level. Usually, safety systems coexist with control systems which implement the main control functionality of the system under control. Both safety and control systems control the same system/process. Thus the control system needs to be taken into account when a safety system is developed.

Development of functional safety systems is heavily regulated by legislations, such as the European Machinery Directive [2]. The legislation refers to standards in which requirements for the development process, techniques, and methods to be used in functional safety system development are documented. However, standards and legislation provide few practical solutions to the development of safety systems especially in context of safety and control system cooperation. The purpose of the patterns illustrated in this paper is to provide solution models for this problem area.

The patterns presented here are part of a larger collection of patterns. Six of the patterns of the collection have been published in VikingPLoP 2012 [5]. The patterns of this article partly relate to the formerly published patterns. We provide short descriptions (patlets) of the referred patterns in this article.

We began the pattern work in autumn 2011 as a result of a software safety related research project. The patterns are not directly discovered from real systems or applications. Instead, our approach is constructive: the patterns are sketched and documented based on our vision of a potential pattern and information gathered from standards related to safety system development, literature, and discussions. Finally, the patterns are (if possible) discussed with industry professionals to gain confidence on the solutions and approaches used in the patterns. Our intention is, however, to identify real world applications and sources for all the patterns. We have discussed the patterns of this paper with industry members from three different companies. The discussion provided support especially for the first two patterns of this paper.

### 1.1 Pattern overview

The patterns belong to a larger pattern collection, which consists currently of some forty patterns or early pattern drafts and ideas. The patterns are related to each other to some extent but cannot yet be considered as a pattern language. To produce a pattern language out of the collection focusing on the patterns and research for missing links would be required. A pattern language is our goal, but to achieve that large amount of work is required.

The current root pattern of the collection is the SEPARATED SAFETY pattern [5], which justifies separation between safety and control systems. The relations of the patterns are depicted in **Fig. 1**. The semantics of relations is as follows. A one way solid arrow from X to Z is: the pattern Z can be applied or considered after application of pattern X. The override patterns box illustrates three alternative solutions for control system override. The patterns presented in this article are highlighted.

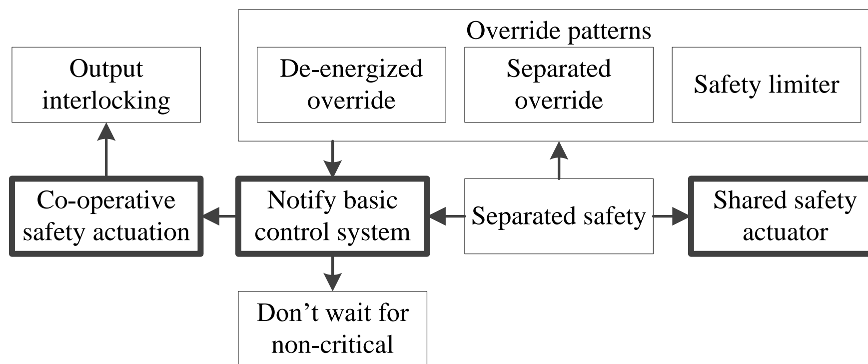



**Fig. 1.** Relations of the patterns

The short descriptions of the patterns referred to but not discussed in this article, are given in **Table 1**.

**Table 1.** Pattern descriptions

| Pattern | Description |
|---|---|
| Separated safety [5] | Development of a complete system according to safety regulations is a bureaucratic and slow process. Therefore, divide the system into control and safety systems and develop only the safety system according to safety regulations. |
| De-energized override [5] | A safety system must be able to override a control system whenever the systems control same process quantities. Therefore, let the safety system use de-energization of the control system's actuator(s) to obtain a safe state. |
| Safety limiter [5] | A safety system must be able to override a control system whenever the systems control same process quantities. Therefore, disengage the control system completely from the actuator and let the safety system |

| | |
|---|---|
| | control the actuator. Route the output of the control system to the safety system and let the safety system treat the control value so that safe operation is ensured. |
| Separated override [5] | A safety system must be able to override a control system whenever the systems control same process quantities. Therefore, provide the safety system with separate actuator to obtain safe state. |
| Output interlocking | A control system must protect machinery, environment and humans from being damaged. Implementing protective interlocking functions in control algorithms makes the algorithms complex and hinders the reusability of the algorithms. Therefore, use an interlock element alongside each control actuator output in the control system and implement the interlock algorithm in these elements. |
| Don't wait for non-critical | A failure of a control system may cause a blocking of the safety system when data is communicated between the systems. Therefore, use asynchronous messaging to arrange communication from the safety system to the control system and do not let safety system wait for a response from the control system (eternally). |

# 2 Patterns

In this section three patterns for functional safety system development and structure is presented. The patterns are presented in the canonical form.

## 2.1 Control system notification

**Context**

The SEPARATED SAFETY pattern has been applied, so safety and control systems are separated. The safety system is capable to control/affect one or more process variables[1] that are also controlled or used by the control system. For instance, safety system can affect to the state of steam flow in pipeline that is used by a control system to regulate temperature of container. To ensure safety, the safety system is able to override the control system regardless of the state of it (e.g. SEPARATED OVERRIDE, DE-ENERGIZED OVERRIDE, or SAFETY LIMITER pattern have been applied). The context is illustrated in **Fig. 2**.

[1] Process variable illustrates current state of (a part of) a process/system typically variable over time. For example, a process variable can illustrate pressure of fluid in pipeline (e.g. the pipeline after hydraulic pump and before the decompression valve). In electric circuits a process variable could illustrate the potential, i.e. the voltage, of certain circuit node or current through a branch.

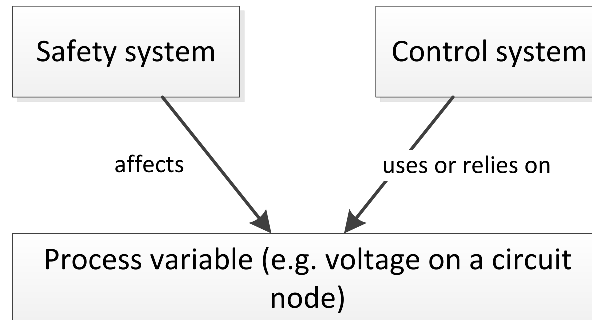



**Fig. 2.** Context of the Control system notification

**Problem**

The operation of the control system is disturbed when the safety system overrides or restricts the operation of the control system, which may cause unexpected behavior of the control system.

**Forces**

- A safety system needs full control over the process variable regardless possible side effects on the control system
- A control system cannot operate normally if safety system has restricted its environment. For example, control system that uses electric current to regulate an electric motor speed cannot operate correctly when a safety system has cut off the current.
- Providing the control system with required hardware to sense the state of the safety systemincreases the cost and complexity of the control system

**Solution**

Make the control system aware of the state changes of the safety system so that the control system can react accordingly. Notify the control system about any operation or event that affects the operation of the control system. Such events are for instance:

- a restriction of a variable, such as limited speed or load
- force control of variable to fully enabled or disabled, such as fully closed steam supply
- return from safe (or restricted) state after a hazardous situation

Implement the notification system so that the safety system is kept as independent of the control system as possible to prevent a blocking of the safety system due to the failures of the control system. Three approaches to achieve such information transfer mechanism can be identified:

- Analogue signaling: The safety system provides an analogue signal for the control system. This approach is simple and releases developers from considerations of additional requirements of the IEC 61508-3 [4] for digital message busses. Howev-

er, analogue signaling requires a dedicated cable between the systems and is more prone to interference from the environment.
- Message bus: The safety system and the control system communicate through a message bus. Safe communication is established through the bus. Additional requirements as given in IEC 61508-3 [4] need to be considered.
- Integrated control and safety system: The safety and the control system are executed in the same integrated device. The device and underlying operating system provides communication scheme between the entities. However, in such a mixed criticality system there has to be separation between the systems in spatial and temporal domains [4].

Regardless the method to pass safety system state and event information to control system, both systems are added with complexity. A communication method between the systems needs to be established, which may increase hardware requirements, unless the communication method already exists. In any way the amount of logic in safety and control system side increases. To enable communication and successful reaction the safety system has to produce the state and event information for the control system and the control system has to receive and use the information in meaningful way. This adds requirements of both system and increases complexity.

**Consequences**

+ The control system can react and adapt to state changes and actions of the safety system
+ Increased overall system safety by decreasing the likelihood of unexpected behaviour related to inconsistencies between safety and control systems
− Full separation between the safety system and control systems is lost
− Increases complexity of both safety and control systems, a notification must be produced and transferred by the safety system and received, interpreted and reacted on by the control system.
− Increased complexity may introduce new programming errors and add latency

**Example**

Consider a simplified heating system illustrated in **Fig. 3.** Steam supply in a heat exchanger is controlled with a proportional valve for temperature control purposes. Steam supply is limited (on-off) by a safety system. When the safety valve is open, the control system controls the steam flow. When the safety system detects a hazardous situation (e.g. high temperature in temperature controlled tank), it closes the safety valve thus blocking the steam flow. If the control system is not notified about the supply cut-off, it keeps trying to control the temperature. Because there is no steam supply, temperature decreases and controller opens the control valve completely and keeps it open trying to increase the steam flow which is not available. This is typically undesired behavior especially when steam supply is allowed again.

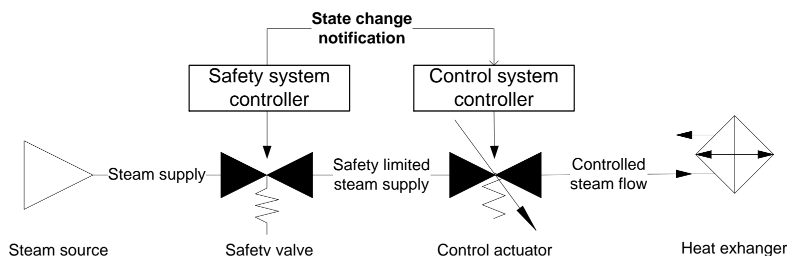



**Fig. 3.** Safety system notification in heating process.

If the control system is notified, when the steam supply is cut off, it can react accordingly and, for instance, close the control actuator (see the CO-OPERATIVE SAFETY ACTUATION) and halt (and reset) the control algorithm to prevent the saturation of an integrator.

**Related patterns**

The CO-OPERATIVE SAFETY ACTUATION pattern describes an application for the notifications from the safety system to the control system. Consistency between the safety and control system can be increased by forcing the control systems actuators to actuate the safe state [1].

The DON'T WAIT FOR NON-CRITICAL pattern suggests the usage of asynchronous communication scheme between safety and control systems to increase isolation between the systems and prevent a blocking of the safety system due to a failure in the control system.

## 2.2 Co-operative safety actuation

**Context**

An actuator of a control system affects a process variable related to a safety function operation. The CONTROL SYSTEM NOTIFICATION pattern has been applied to enable safety system notifications from the safety to the control system.

**Problem**

How to increase consistency between operation of safety and control system during situations in which the safety system overrides partly or completely the control system?

**Forces**

- Conflicting state between safety and control system may cause undesired operation of the control system and increase risk of malfunctioning safety function
- Consistent state of safety and control systems regarding the process variable affected by both systems increases the reliability of successful actuation of the safety function because in case of either actuator fails the other may still be able to actuate the correct safety function result

**Solution**

Let the safety system drive the control system into a safe state whenever safe state needs to be obtained (according to the safety system). That is, the state information of the safety system is actively used as an input in the control system. As the safety system can already notify the control system about the state of the safety system, it is relatively simple to go further and use this asset to increase consistency between the functionality of the systems. Consistency between the states of the control and safety systems decreases the state space in which the system can be when a safety function is active (after transitions).

When the actuators of the safety and control systems are in a consistent state, the reliability of successful outcome of a safety function is increased. The actuators of the control system are primarily used to control the process/system to produce a desired output, but they affect the state of the system similarly to the actuators of the safety system, i.e., change the state of the process variables (e.g. fluid flow or electric current). Thus, the actuators of the control system can be used to actuate similar operations as safety functions. However, they can only support the safety system, not take its responsibilities (in which case the control system turns into a safety system).

A possible way to implement the pattern is to command the control system into a safe state by a notification. That is, the safety system sends a message/notification to the control system that processes the message and reacts accordingly by driving the actuators (under its control) to safe states. In the simplest case safety system only commands the control system into the predefined safe state. The approach is illustrated in **Fig. 4** in which the Safety System first closes the Safety Actuator and then notifies the Control System to take the safe state (which is assumed to be predefined). The control system receives the notification and drives the Control Actuator into the safe state. In more advanced cases, the safety system communicates the desired state (e.g. valve_1 = closed, valve_3 = open) and the control system drives the actuators into these states.

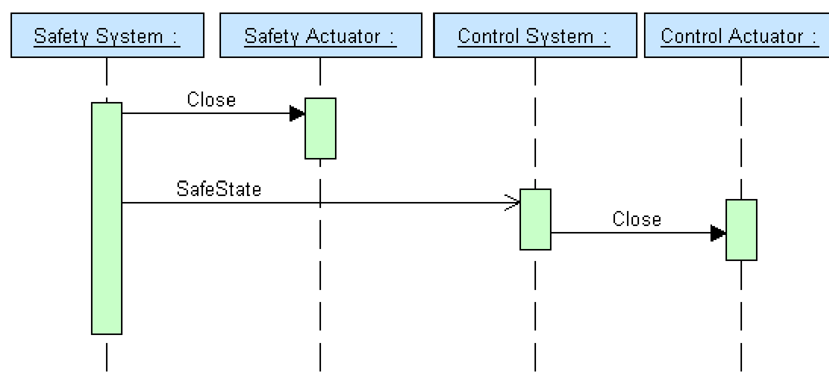



**Fig. 4.** Co-operative safety actuation through control system notification

**Consequences**

+ Increased consistency between the states of the safety and control systems which decreases possibility of malfunctions due to state inconsistencies (when giving control back to the control system)
+ Increased reliability of the desired outcome of safety function actuation due to redundancy in actuators, but…
− …the control system side provides only additional "peace of mind" reliability which cannot be counted into safety system attribute as such
− Increased complexity of control system algorithm as it needs to take the safety system input into account
− May mask safety system actuator faults (unless diagnosed otherwise)

**Example**

Consider the simplified heating system illustrated in **Fig. 5** The safety system can notify the control system. Now, when the safety system triggers the safety function the safety valve is closed. To improve consistency between the systems and reliability of steam cut-off the safety system informs the control system about the safety function and requests it to close the control actuator as well. The control system is in a consistent state with the safety system and does not even try to continue normal control operations as it would be impossible to operate successful control under safe state circumstances.

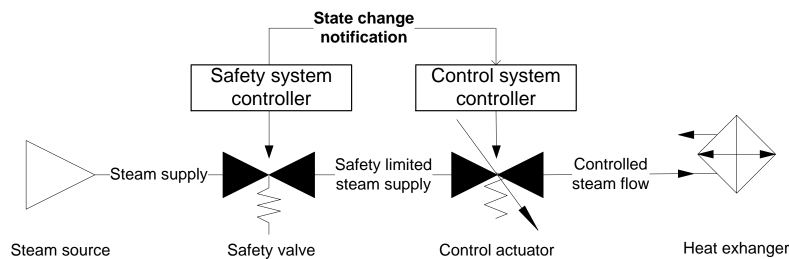



**Fig. 5.** Safety system notification in heating process.

The control actuator is of a proportional type, which is not typically used in safety systems. The proportional valve may e.g. leak more than a simple binary valve. However, it is better to have a marginally leaking proportional valve closed than a faulty safety valve fully open.

**Related patterns**

The OUTPUT INTERLOCKING pattern suggests the usage of interlocking elements alongside each control output used to control a physical device. These interlocking elements provide a way to implement the driving of the control system into a suitable state. That is, when an interlock of control system receives request to obtain safe state, it forces the control output predefined safe state.

The DE-ENERGIZED OVERRIDE pattern describes a potential way to implement the cooperative actuation. In this approach, the safety system has full and direct control to drive the actuator of the control system into a safe state by de-energizing the control actuator(s).

## 2.3 Shared safety actuator

### Context

A system under control consists of subsystems that use an input produced by a single source as illustrated in **Fig. 6** (typically the source is an energy source). A similar safety function is related to all the subsystems (e.g., an emergency stop). The safety function operates in the same direction and has the same safe state in terms of the shared input between the subsystems. That is, each subsystem takes safe state, e.g., when the input is disconnected from the subsystem. The SEPARATED OVERRIDE pattern has been applied, i.e., the safety system has a dedicated actuator to control operation of the system.

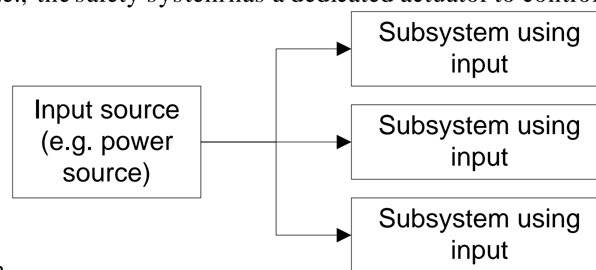



**Fig. 6.** The context of the shared safety actuator

### Problem

Providing each subsystem with dedicated safety actuator when same input variable is used by multiple subsystems increases the amount of needed safety actuators in the system.

### Forces

- Dedicated safety actuators for each subsystem does not decrease productivity, flexibility and availability of the system by letting each independent subsystem continue operation in case one of the subsystems needs to obtain safe state
- Dedicated safety actuators for each subsystem increase hardware cost, weight, space requirements and complexity of the safety system
- Suitable safety actuators are considerably expensive or there are space and/or weight requirements considering the actuator and thus the number of the actuators is wanted to be kept low
- All the subsystems share common safe state in terms of the considered input
- Independent operability of the subsystems, in terms of the shared input, can be sacrificed for other attributes (e.g., cost and weight)

**Solution**

Use a shared safety actuator for all the subsystems. The safety actuator is positioned so that it can control the safety function considering all the subsystems (in context of the shared input). The principle of the solution is depicted in **Fig. 7**. In the figure, a safety actuator is added between the input source and the subsystems which use the input and which are safety-critical. The safety actuator controls the input. Whenever the safety function (related to the input) is trigged in any of the subsystems, the safety actuator is used to obtain a safe state. The safe state propagates to all subsystems regardless of their state.

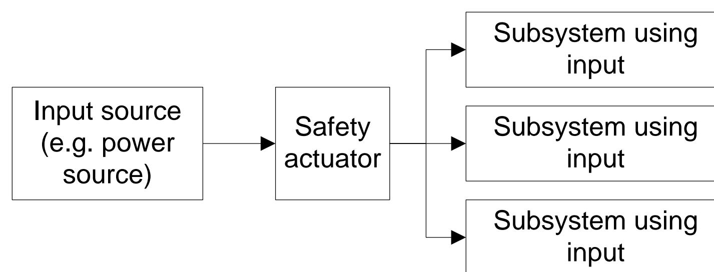



**Fig. 7.** Shared safety actuator principle

Ensure the decoupling of the subsystem in terms of the shared input. That is, substance that has once entered in a subsystem cannot move to another subsystem without first circulating through the input source. The decoupling should especially be considered when the shared input is an energy source of some kind and the subsystems store energy (see example). If the decoupling devices are critical part of the safety function, they should also be considered as part of the safety system.

The solution requires thorough consideration before application. There are many aspects that might result in problems. Subsystem decoupling is one, but also the topology, structure and functions for exceptional operation may cause undesired side-effects when the approach is used. In this solution only the main approach is presented. Actual application of the solution depends on the details of the target system. For instance, in a hydraulic lifting system one should consider potential energy stored in a lifted object and prevent object movement and drifting.

The input source is typically a power/energy source of some kind such as hydraulic (hydraulic motor), electric (power source) or pneumatic (air compressor) energy source. However, the input source can be any controlled variable of the system. The input source itself can also act as a “safety actuator”. That is, the whole source of the considered variable can be turned on or off controlled by the safety system. This, as well, requires a thorough analysis of the effects to prevent undesired functionality.

**Consequences**

+ Decreased amount of safety actuation hardware
+ Decreased space, weight, and (potentially power consumption) requirements for a safety system due to lower amount of safety actuation hardware

+ Potentially decreased overall cost of the safety actuator hardware
− Potentially decreased productivity, flexibility and availability of the system because the subsystems (sharing the input) lose independency considering safety function(s) related to the shared input
− The potential safety functions are (practically) restricted to on-off type, because it is hard arrange distribution of the shared input between the subsystems. In practice this would need additional hardware which hinders the original objective of the approach.
− The subsystem design may see new requirement to meet the requirement for correct operation of the safety system
− Shared actuator requires dedicated control element (e.g., a software component that is responsible for the actuator control)
− The approach is prone to unpredictable side-effects, due to, for example, insufficient decoupling between subsystemin terms of the shared input
− Requires detailed and throughout analysis to ensure correct operation in various operational cases
− Safety system has to be developed for the highest criticality level of the subsystems

**Example**

Let us consider a harvester machine that has a hydraulic boom. To reduce the harvester's weight and power consumption all the boom cylinders share a safety actuator that is able to halt the boom movement. A block diagram of the system is provided in **Fig. 8**. On the left hand side there is a hydraulic pump that represents the input source, i.e., hydraulic power in this case. After the pump there is a safety valve that controls the flow to the boom cylinders. On the right there are the actual boom cylinder control valves that are controlled by the control system.

An important detail in the presented schematic is the check valves that decouple the cylinders (i.e., subsystems using the shared input) from each other. The check valves ensure the hydraulic fluid and pressure cannot transfer directly from cylinder to another, but rather has to circumvent through the tank.

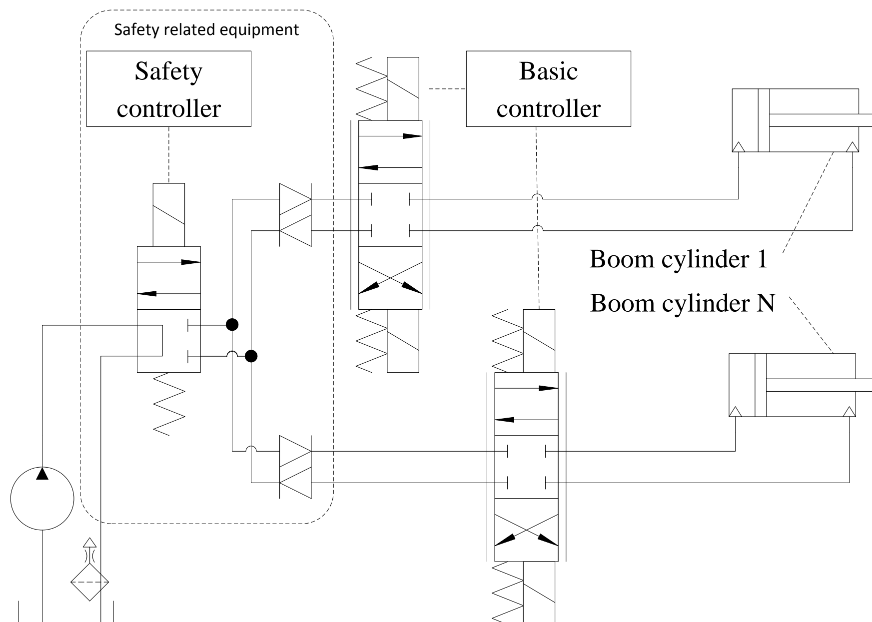



**Fig. 8.** Shared safety actuator for harvester boom cylinders

**Known use**

The BGIA Report 2/2008 (section 8.2.27) [3] illustrates similar solution approach. However, the approach given in the report focuses on redundancy and employs safety related actuators also in the subsystems to avoid single point of failure problem. The decoupling aspect is not considered. Nevertheless, it is mentioned that the shared safety actuator is sufficient to enable the considered safety stop safety function.

**Related patterns**

The safety system should notify the control systems of the related subsystems as illustrated in the CONTROL SYSTEM NOTIFICATION.

# 3 Acknowledgements

The authors would like to thank our shepherd Dirk Schnelle-Walka and all the industry members for their valuable input. Our thanks also belong to the participants of the VikingPLoP 2013, whose comments greatly improved the paper.

# 4 References


1. V. Eloranta, J. Koskinen, M. Leppänen, and V. Reijonen, A Pattern Language for Distributed Machine Control Systems, 2010. http://practise.cs.tut.fi/project.php?project=sulake [retrieved: February, 2013].

2. European Parliament and of the Council, Directive 2006/42/EC of the european parliament and of the council, vol. L 157/24, 2006.

3. Hauke, M., Schaefer, M., Apfeld, R., Bömer, T., Huelke, M., Borowski, T., Büllesbach, K.-H., Dorra, M., Foermer-Schaefer, H.-G., Grigulewitsch, W., Heimann, K.-D., Köhler, B., Krauß, M., Kühlem, W., Lohmaier, O., Meffert, K., Pilger, J., Reuß, G., Schuster, U. & Zilligen, H. Functional safety of machine controls - Application of EN ISO 13849. BGIA Report 2/2008e, 2008. 373 p.

4. International Electrotechnical Commission, Functional safety of electrical/electronic/programmable electronic safety-related systems, Part 3: Software requirements, IEC, 2010.

5. Rauhamäki, J., Vepsäläinen, T., Kuikka, S. "Functional Safety System Patterns," Proc. VikingPLoP 2012 Conference, 2012, pp. 48-68, http://URN.fi/URN:ISBN:978-952-15-2944-3 [retrieved: February, 2013].

# A pattern for bootstrapping

Ville Reijonen

Kauppalehti Oy / Tampere University of Technology
{firstname.lastname}@tut.fi

## 1 Bootstrapper

...there is a Control System which consists of hardware parts and an application(s). Hardware has to be prepared before the application can be given the reins. For example, memory has made to be activated and initialized. Additionally, no hardware is immune to errors, faults or decay. Unexpected issues may emerge if application is run on faulty hardware. For the application it would be preferable if the environment would be in a defined condition after every start-up sequence.

**Before application can be started the hardware should be in defined state. Otherwise fault of uninitialized hardware may cause unexpected behaviour.**

After power up, the basic functionality of the hardware has to be activated with right parameters. For example, basic memory and internal buses need to be activated. If these are not set up, or are set up incorrectly, it is likely that the system does not work.

Usually there is only limited amount of memory and persistent storage available when the system powers up. Program code required for activating all the hardware might not fit in the memory or storage available during power up. Additional memory and persistent storage might be available only after they have been activated and set up.

There are certain diagnostics and self-tests, such as memory error check and bus connection checks that should be done before commencing with the actual task of the device. A self-test operates at the lowest level possible with the hardware. Creat-

ing such tests typically requires deep knowledge on the actual hardware. Thus, it should not be application programmer's task to make these self-tests and checks.

The application might be stored on variety of different medium. As it cannot always be known beforehand on which connected media the application resides, a way to discover it may be required. There could be multiple alternative applications in the system, or even alternative versions of the same application. In that case a way to select the loaded application is needed.

System might be created by bundling together COTS hardware from multiple vendors and adding in-house hardware on the top of that. Thus COTS hardware cannot initialize all hardware and hardware combinations in the system. It is on the responsibility of the system manufacturer to integrate and create initialization for the whole.

**Therefore:**

**Initialize the hardware during start-up so that the system is always in consistent state during start-up. Divide the start-up to sequential stages, when necessary, to overcome the system resource limits and for separation of concerns [1]. For each stage add a bootloader component with its own responsibilities.**

Each bootloading stage adds flexibility to the boot process. As an example of overcoming resource limits, the persistent memory available during start-up is often limited in size and therefore the first stage bootloader can only contain the very essential initializations. The foundations for the second stage are created in the first stage by setting up storage device, initializing memory and loading second stage to the memory. As an example of the separation of concerns, a CPU board manufacturer does not know where their board is going to be used. Therefore the CPU board only takes care of its own setup. Additional stages are required to handle the system beyond the CPU board.

The first stage after power-up is called bootstrap loading or boot loading. Here the basic functionality of the hardware is initialized. The bootstrapper code resides on limited sized persistent storage such as ROM or EEPROM, available immediately during start-up. A CPU wakes up and executes program code from defined memory address on this persistent storage. One of the first tasks for the bootstrapper is to activate different kinds of buses in the system such as memory bus, data bus and control bus. Proper bus timings have to be used to ensure correct behaviour, otherwise data loss and undefined errors will occur. When the buses have been activated the volatile memory, RAM, can be activated. Additionally, storage device may be activated along with some basic peripheral devices such as serial ports. This is the minimum what a bootstrapper should do to enable loading latter stages or running an application.

In addition, the first stage may contain power on self-tests (POST) for processor, memory, controllers, system buses and other basic hardware. POST helps to detect hardware errors, both during software development and in use. During software development hardware bugs are often time consuming to diagnose and treat, therefore hardware should be as stable as possible. The level of use time testing depends on the requirements for the system. For example, undetected hardware error in a satellite might lead to unrecoverable situation. After POST phase additional bootloader stage may be loaded to the memory and executed. Latter stages may contain their own set of POST tests for those parts of hardware which they have enabled. The hardware manufacturer often provides the first stage bootloader as it requires extensive proprietary knowledge on hardware internals which they only have.

Often the location of the second bootloader is fixed, that is, it is expected to reside in certain location on activated persistent storage. This makes the task of loading the second stage easier and it requires fewer resources from the first stage bootloader. As the second stage code resides on larger medium and there is more volatile memory available, it can do much more that the first stage. Typically the second stage bootloader enables additional devices. Such a device could be, for example, advanced storage and memory solutions, networking and wireless interfaces and peripheral devices such as keyboards and displays. Sometimes the second stage functions just as a stepping-stone to third stage. For example, the second stage could have functionality to search all storage media such as SD cards, USB sticks or hard drives for the third stage. In such a design the second stage is more like extension to first stage.

In some embedded systems latter stages are not required or wanted due to timing concerns. For example, safety systems in a power plant do have high requirements for availability. Loading the main application as soon as possible requires that the hardware is tightly coupled with the software, so that additional setup or testing is not required. Device initialization is usually fast process, but a lot of time might be spent on some self-tests such as extensive memory tests. Consequently, the tests have been sometimes divided into two groups; fast power on self-tests and extensive long time running diagnostic tests. Even when fast timing is not crucial, the users should not kept waiting for no good reason. Therefore, it is not often reasonable to run long lasting tests all the time.

More advanced bootloader may contain more functionality such as user interface, logic for selecting and booting alternative applications, backup service, software updating functionality such as described in Updateable Software pattern, rescue mechanisms or rescue mode, etc. If the bootloader has more than one application which it could execute, there should be a way to select which one to use. There can be multiple reasons for having multiple application such as to provide a way to use different versions of the software, have separate rescue mode application, bundle extensive diagnostic test as an application or have higher availability by applying 1+1 Redundancy pattern. The selection mechanism can be implemented with software logic or with hardware switch.

One typical rescue mechanism is bootloop detection. In a bootloop system crash is detected by Watchdog, which reboots the system to have it crash again and again. One way to implement bootloop detection is to have a counter which increases on start-up and resets on shutdown. If the counter value is larger than zero during start-up, the system was not shutdown cleanly. When the value is larger than one, the system has crashed multiple times in row and may be in bootloop. In this case the operator of the system is alerted, alternative version of the application could be tried or system might be halted.

Sometimes third parties should not be allowed to tamper with the system software. Way to secure the bootloader and system from modification and external tampering is presented in SECURE BOOT pattern by Hans Löhr et al. [2]. The basic idea is to use checksums or cryptographically signed bootloaders and applications. Only software with valid checksum or signature is loaded and executed. Part of the verification chain is done in hardware. This makes it more secure as it is harder to modify or read out hardware than software. Such a solution is used for example in SIM cards, electronic cash cards, game consoles and some mobile phones.

A BOOT LOADER pattern by Dietmar Schuetz [3] descibes the problem area and solution more from the x86 PC and hardware perspective. Many of the ideas presented in the pattern still apply any hardware or CPU setups even if the hardware details differ.

Bootloading ensures that the system is functional with required components. The POST verifies that the system is in tested defined state. This creates a good stable base, which can be relied upon. Application developer does not need to wonder if the hardware is in working order. However, as everything cannot be tested, testing just makes sure that the probable and common errors are detected. With dublication and rescue mode additional availability may be achieved.

When the bootloader is divided into stages, every stage provides always higher level of service on top of the previous stage. Typically the hardware vendor concentrates on the hardware details in their first stage bootloader and company using the hardware adds functionality of their own into latter stage bootloader(s).

Especially the first stage bootloader is usually highly hardware dependent and creating such a piece of software requires good knowledge on the hardware and how to program it. Typically hardware registers are manipulated and low level coding language such as assembler is used in some parts of the bootloader. This requires detailed knowledge and skill set, which an normal application developer rarely has.

If system initialization is build using multiple bootloaders, the startup is not as fast as it could be. In many case delayed startup sequence is acceptable cost for the re-

ceived flexibility. When a system should serve its task almost immediately after powering up it is not acceptable to wait. Too long start-up period is also usability issue from the operator perspective.

When the ARM CPU is powered on, its registers are set to predefined values. The processor starts to execute binary from address 0x00000000. This memory address resides in internal ROM memory located on-chip with the CPU. The first stage bootloader is preloaded by hardware manufacturer into internal ROM of the CPU. The bootloader initializes the system by setting busses, clocks, stacks, interrupts etc. After this, the bootloader identifies the boot media by looking for bootloader signature on external flash and then from USB storage.

If the second stage bootloader is not detected, the first stage bootloader halts the system. If bootloader is found from medium, it is loaded into internal RAM. Then the CPU program counter is set to address of the second stage bootloader load address. The second stage bootloader is responsible for loading operating system into memory. For this to be possible, the memory has to initialized first by setting up controller, memory refresh rate, etc. The second stage bootloader has been configured to launch operating systems from certain locations defined beforehand. After the memory has been initialized, the first location is tried. If no operating system is found, next address is tried and so on. If no operating system is found, the system is halted. When suitable signature is found, the operating system is loaded into memory and execution is transferred to the operating system.

## 2 Acknowledgements

I would like to thank all the participants of the VikingPLoP 2013 for their input on this work.

## 3 References

1. Hürsch, W., Lopes, C.: Separation of Concerns. Technical report by the College of Computer Science, Northeastern University. 1995.
2. Schuetz , D.: Boot Loader. In proceedings of 11th European Conference on Pattern Languages of Programs, 2006.
3. Lohr, H., Sadeghi, A-R., Winandy, M.: Patterns for Secure Boot and Secure Storage in Computer Systems. In proceedings of ARES'10 International Conference on Availability, Reliability, and Security, pp. 569-573. IEEE, 2010.

# Probabilistic Dialog Management

Dirk Schnelle-Walka, Stefan Radomski, Stephan Radeck-Arneth
Telecooperation Group
Darmstadt University of Technology
Hochschulstraße 10
D-64283 Darmstadt, Germany
[dirk|radomski|stephan.radeck-arneth]@tk.informatik.tu-darmstadt.de
phone: +49 (6151) 16-64231

October 11, 2013

**Abstract**

Modeling user interfaces as dialogs provides a conceptual framework to address global coherence and efficiency of interactions. While non-probabilistic approaches provide convincing results and transparent dialog behavior, probabilistic techniques can help to account for inherent uncertainties in user input. In this paper, we present three patterns for probabilistic dialog management or support thereof.

## 1 Introduction

Describing graphical user interfaces is still, predominantly, achieved by applying the Model-View-Controller pattern [5] or one of its variations. Wherein the graphical widgets of an application provide the means to input data as the view, processed by a controller component to adapt the model of an application, which in turn, updates the view. This pattern works very well for graphical user interfaces in the absence of recognition errors or with inexpensive error correction.

There are two problems with the MVC pattern with regard to generic user interfaces. First, it does not ensure a coherent global dialog behavior between the user and the system. Second, it assumes unambiguous user input, recognized without any errors, which is not the case for interfaces employing spoken or gesture input. Therefore, we identified MVC as an anti-pattern [6] as far as dialog management is concerned.

This paper is aimed at developers of multimodal interfaces and extends our pattern language introduced in the aforementioned earlier paper. We describe patterns to support global dialog coherence by probabilistic approaches, accounting for inherent recognition uncertainty with some modalities.

## 2 Patterns

The patterns described in this paper integrate into the language of patterns for dialog management that we introduced in [6]. An overview of the pattern language and their relations is shown in figure 2. The earlier patterns

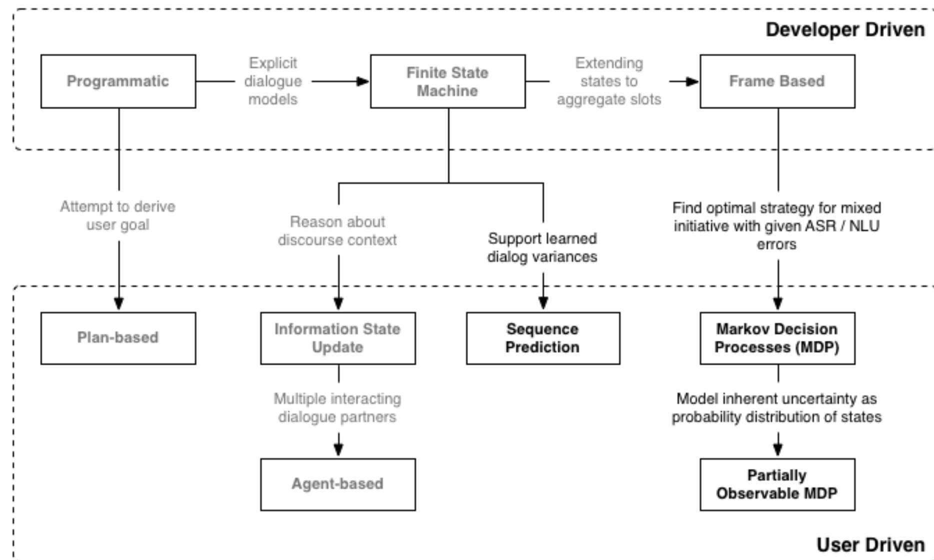



Figure 1: Overview of the pattern language

of our language are shown as gray. Table 2 summarizes these patterns as (External) Pattern Thumbnails [3].

In this paper we extend our language by three more patterns: Markov Decision Processes (MDP), Partially Observable MDP and Sequence Prediction. While the first two have been established in research around dialogue management in the past decade, the latter is more of an explorative nature. This means that we use the pattern format to explore the domain. We aim for extending existing research in sequence prediction algorithms mainly discussed in [2] to discuss their applicability to dialog management. Consequently, there are no known uses.

The following terms are defined in more detail in our original paper [6] and only included here for completeness.

**Dialog (-strategy)**: A recursive sequence of inputs and outputs necessary to achieve a goal [4].

**Dialog Turn**: A single input or output within a dialog.

**(Information) Slot**: Storage to hold a single atomic piece of information.

| Name | Intent |
|---|---|
| Programmatic Dialog Management | Implement an interactive application with unimodal interaction and no need for explicit dialog management. |
| Finite State Dialog Management | Provide an interactive application with an easy way to adapt the dialog structure later on. |
| Frame Based Dialog Management | Allow for adaptations of dialog structure without altering application logic, but try to ease the verbosity of finite state dialog models. |
| Information State Update | Allow for more flexible dialogs with a certain amount of *intelligence* in the dialog structure. |
| Plan Based Dialog Management | Uncover the user's underlying goal to guide the actual dialog management. |
| Agent Based Dialog Management | Model interaction with distinct subsystems as agents with their own beliefs, desires, intentions (and obligations). |

Table 1: Thumbnails of patterns described in [6]

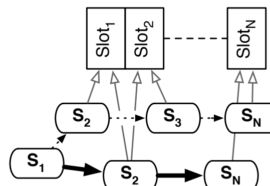
**Mixed Initiative**: The possibility for a user to provide compound information during her turn, as opposed to a single information; Fills multiple information slots at once.

Patterns are in a custom format which is based on the Coplien format [1] which we find useful to talk about human computer interaction.

## Markov Decision Processes (MDP)

### Intent

Find an *optimal* dialog strategy to fill a set of information slots with a sequence of mixed-initiative dialog turns, where a single user action can potentially fill multiple slots.




### Context

A mixed-initiative dialog needs to sufficiently instantiate a template as a set of information slots. Different (compound) user actions yield different recognition accuracies depending on their complexity and the performance of a recognizer and an eventual natural language understanding unit. Frame Based Dialog Management can be applied to allow for an arbitrary order to fill the slots but does not take the different recognition rates with compound user actions into account.

### Problem

There are multiple, possible sequences of state transitions, each state prompting the user to perform an action that will potentially fill multiple information slots. The recognition accuracies differ with regard to the amount and kind of information the user action contains for a single prompt. How to account for those different recognition accuracies to find an optimal and consistent dialog strategy to sufficiently instantiate a template?

### Forces

- A sufficiently large corpus of example dialogs exist or users can be simulated.
- Dialog is limited in scope to prevent state explosion.
- The goal of the user can be conceived as the instantiation of a template.

### Solution

The solution tries to find the optimal dialog strategy to instantiate templates as a set of information slots as defined by an objective cost / reward function [3]. By modeling the dialog as a Markov Decision Process (MDP), approaches from reinforced learning can be applied to minimize these cost / reward function with regard to the dialog turns. Paek and Chickering analyzed in [4] different reward models and their suitability to constrain the state space when its structure is unknown.

A MDP is formally defined as the quadruple:

$$
\begin{aligned}
MDP &:= \quad S, A, T, R \\
S_{tates} &:= \quad \{s \in \mathit{all\ dialog\ states}\}
\end{aligned}
$$

$$
\begin{aligned}
A_{ctions} &:= \{a \in \textit{all output actions}\} \\
T_{ransitions} &:= P(s_{t+1}|s_t, a) \\
R_{eward} &:= r(s, a)
\end{aligned}
$$

With $S_{tates}$ being the cartesian product of all the information slots with their possible values. $A_{ctions}$ as a set of dialog acts the machine can perform to prompt the user to provide input, $T_{ransitions}$ as the probabilities to transition from state $s_t$ to $s_{t+1}$ should the action $a$ be selected and $R_{eward}$ as a cost / reward function, associating a cost for performing an action in a given state.

The dialog starts in the state where all information slots are unfilled. The transition with the highest probability is taken and the associated action is performed (e.g. open prompt greeting). The user's input is processed by a semantic interpretation unit and the new state determined. The process continues until one template is sufficiently instantiated for the system to satisfy the users goal. To apply a Markov Decision Process for dialog modeling, consider the following:

1. Establish the state-set $S_{tates}$ as the cartesian product of all possible input field values.

2. Identify all possible actions $A_{ctions}$ that are relevant to perform the dialog.

3. Provide a cost function to account for e.g. overall dialog length, cost of database queries, number of unfilled information slots, cost of rendering a new prompt.

4. Use reinforced learning to train the $T_{ransition}$ probabilities from the dialog corpus or user simulation.

The resulting MDP provides the basis for a dialog strategy, satisfying the optimality criterion implicit in the cost function.

## Consequences

- ☺ Resulting dialog strategy is learned from real data.
- ☺ All dialog strategies are (nearly) optimal with regard to cost function.
- ☺ Accuracy of recognition and language understanding is taken into account.
- ☹ The cartesian product of all possible input field values tends to be huge and an optimal solution often intractable.
- ☹ Actual dialog behavior is opaque as it is encoded in the MDP.
- ☹ Adapting the dialog requires retraining.

☹ Recovery strategies, to realign the users interaction intent with the systems interpretation [5], are difficult to implement, as all the different approaches would need to be formalized in the cost function.

### Known Uses

Lemon showed in [2] that dialog management and natural language generation are closely related and that a joint and automated training result in a significantly better reward.

Boyer et al. introduce in [1] a tutorial dialog system based on this pattern.

### Related Patterns

Markov Decision Processes (MDP) extends Frame Based Dialog Management to find an *optimal* dialog strategy to fill a set of information slots with a sequence of mixed-initiative dialog taking into account the respective recognition rates for such compound input.

Partially Observable MDP helps to model this uncertainty as a probability distribution of states.

### References

[1] Kristy Elizabeth Boyer, Eun Young Ha, Robert Phillips, Michael D Wallis, Mladen A Vouk, and James C Lester. Inferring tutorial dialogue structure with hidden markov modeling. In *Proceedings of the Fourth Workshop on Innovative Use of NLP for Building Educational Applications*, pages 19–26. Association for Computational Linguistics, 2009.

[2] Oliver Lemon. Learning what to say and how to say it: Joint optimisation of spoken dialogue management and natural language generation. *Computer Speech & Language*, 25(2):210–221, 2011.

[3] E. Levin, R. Pieraccini, and W. Eckert. Using markov decision process for learning dialogue strategies. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 1, pages 201 –204 vol.1, may 1998.

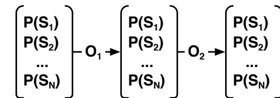
[4] Tim Paek and David Maxwell Chickering. The markov assumption in spoken dialogue management. In *6th SIGDIAL Workshop on Discourse and Dialogue*, 2005.

[5] Dirk Schnelle-Walka. A pattern language for error management in voice user interfaces. In *Proceedings of the 15th European Conference on Pattern Languages of Programs*, EuroPLoP '10, pages 8:1–8:23, New York, NY, USA, 2010. ACM.

## Partially Observable MDP

### Intent

Provide a dialog management system that implicitly copes with the uncertainty related to the recognition of user input.




### Context

Markov Decision Processes (MDP) has been applied to find an *optimal* dialog strategy to fill a set of information slots in a mixed initiative dialog. Still, a huge class of problems, especially with multimodal applications, stems from the fact that user input cannot be recognized with absolute certainty.

### Problem

How to model dialogs with inherent uncertainty in the user input?

### Forces

- User input intend cannot be derived with certainty from employed modalities.
- A sufficiently large corpus of example dialogs exists or user input can be simulated.
- Dialog is limited in scope to prevent state explosion.

### Solution

Implicitly modeling uncertainty as a partially observable Markov decision process with a probability distribution among all the dialog states can help to arrive at concise and effective dialogs in the absence of robust recognition. In order to implement this strategy consider the following:

Model the dialog as a partially observable Markov Decision Process - an extension of the MDP quadruple defined above as:

$$\begin{aligned}
POMDP &:= S, A, T, R, O, Z, \lambda, b_0 \\
S_{tates} &:= \{s \in all\ dialog\ states\} \\
A_{ctions} &:= \{a_m \in all\ output\ actions\} \\
T_{ransitions} &:= P(s_{t+1}|s_t, a_m) \\
R_{eward} &:= r(s, a_m) \\
O_{bservations} &:= \{o \in all\ user\ input\ a_u\} \\
Z &:= P(o_{t+1}|s_{t+1}, a) \\
\lambda &:= geometric\ discount\ factor\ with\ 0 \leq \pi \leq 1 \\
b_0 &:= initial\ state\ probability\ distribution (belief)
\end{aligned}$$

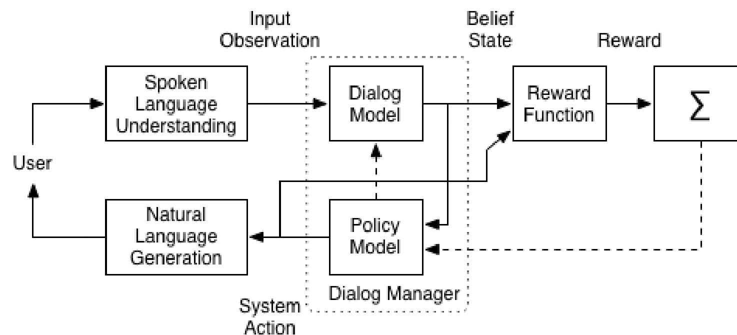
With $S, A, T, R$ as defined in the MDP pattern and $O_{bservations}$ as the set of input actions a user might perform, $Z$ as the probability of observing user action $o_{t+1}$ after performing $a_m$ as part of the previous transition. $\lambda$ as a discounting factor to optionally emphasize late rewards and $b_0$ as the initial state probability distribution.

The major difference when compared to other state-based dialog managers is that POMDPs will maintain a probability distribution of all states in $b$ and employ dynamic programming to determine the most likely user goal and system action [3].

When performing the dialog, the system will receive the users input and perform *belief monitoring* to update the state probability distribution in $b$ for each dialog turn. This distribution gets mapped to actions (e.g. a voice prompt). Several stochastic models are needed in order to operationalize the approach and would need to be trained from a dialog corpus or by user simulation.

As it may be computationally intractable to process the whole belief state and associated actions, Young proposes a grid-based approach [3], where actions are points in the belief state and a distance metric can be employed to find the best action. Another extension is the mapping of all state into a summary state space, containing the most likely states corresponding to user goals.

An overview of the principal components in a POMDP dialog system, based on [3] is shown in the following figure:




## Consequences

- ☺ Resulting dialog strategy is learned from real data.
- ☺ Robust with respect to recognition or understanding errors
- ☺ Implicitly models and takes into account uncertainty in user recognition
- ☹ The state space tends to get huge as it has to model all dialog states and mappings from beliefs to actions.

☹ Users goal is assumed to change infrequently to keep belief monitoring manageable.

☹ Recovery strategies are difficult to implement as they have to be part of the original corpus.

### Known Uses

Wiliams et al. showed in [2] that the performance of this dialog management is comparable to hand-crafted dialog managers.

The Trainbot system [4] uses this dialog manager to make appropriate dialog turns in a given situation.

Tsiakoulis et al. presented in [1] a voice-based in-car system for providing information about local amenities (e.g. restaurants).

### Related Patterns

PARTIALLY OBSERVABLE MDP extends MARKOV DECISION PROCESSES (MDP) by providing the means to model uncertainty as a probability distribution of states.

### References


[1] Pirros Tsiakoulis, M Gašic, Matthew Henderson, J Planells-Lerma, Jorge Prombonas, Blaise Thomson, Kai Yu, Steve Young, and Eli Tzirkel. Statistical methods for building robust spoken dialogue systems in an automobile. In *4th International Conference on Applied Human Factors and Ergonomics*, 2012.

[2] Jason D Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech & Language*, 21(2):393–422, 2007.

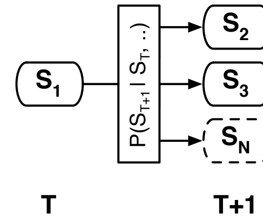
[3] S. Young. Using pomdps for dialog management. In *Spoken Language Technology Workshop, 2006. IEEE*, pages 8 –13, dec. 2006.

[4] Weidong Zhou and Baozong Yuan. Trainbot: A spoken dialog sytem using partially observable markov decision processes. In *Wireless, Mobile and Multimedia Networks (ICWMNN 2010), IET 3rd International Conference on*, pages 381–384. IET, 2010.

## Sequence Prediction

### Intent

Support an actual dialog strategy by making predictions about future user input.




### Context

A state-based multimodal dialog system exists, e.g. Finite State Dialog Management, where 1) multiple strategies can satisfy the users interaction intent or 2) knowledge about potential future steps can be applied to increase the interaction efficiency.

### Problem

How to support the dialog manager with knowledge about previously observed interaction strategies employed by a user?

### Forces

- System- or user-actions can be performed in several ways.
- Users are likely to stick to the approach they identified first [?].
- The system can support the user in an unobtrusive way by using knowledge about probable future input.

### Solution

By learning the observed sequences, a system can provide support to arrive at more concise and effective dialogs. To implement this strategy consider the following:

Establish an N-Gram $P(s_{t+1}|s_t, s_{t-1}..s_{t-N-1}, t(s_t, s_{t+1}) \in T)$ to determine the probability of a state given a history of states and use a sequence prediction algorithms to take a guess at the next state. Support the user by unobtrusively offer short-cuts and interaction support using this knowledge.

1. Count all occurrences of a state in a dialog corpus or online while performing the dialog as the 1-Gram model.
2. Maintain a history and establish the 2..N-Gram models as well.
3. Look-up the N most likely states given the history in the N-Gram and unobtrusively support transitions to these states in the interface.

### Consequences

☺ Can support actual dialog managers with their strategy by hinting at future transitions.

- ☹ Is not suited to perform actual dialog management, but rather a complimentary approach.
- ☹ The N-Grams need to be established per user as their interaction patterns may differ.

An evaluation and overview of available sequence prediction algorithms is available in e.g. [1].

### Related Patterns

Sequence Prediction extends Finite State Dialog Management by supporting an actual dialog manager with predictions about future user input. In contrast to the previous two patterns, this is a complimentary approach to dialog management and not an actual dialog management technique.

### References

[1] Melanie Hartmann and Daniel Schreiber. Prediction algorithms for user actions. In Ingo Brunkhorst, Daniel Krause, and Wassiou Sitou, editors, *15th Workshop on Adaptivity and User Modeling in Interactive Systems*, 2007.

## 3 Conclusion

In this paper we continued the work on our pattern language for dialog management with patterns about probabilistic dialog management. Specifically, we described the following three patterns:

Markov Decision Processes (MDP) extends Frame Based Dialog Management to find an *optimal* dialog strategy to fill a set of information slots with a sequence of mixed-initiative dialog turns with given uncertainty in recognizing the user's actions.

Partially Observable MDP helps to model this uncertainty as a probability distribution of states.

Sequence Prediction is an extension to Finite State Dialog Management by supporting an actual dialog strategy by making predictions about future user input. In contrast to the previous two this is rather a guidance to dialog management.

The major disadvantage of probabilistic approaches, from our point of view, is the opaqueness of the learned dialog strategies, making it hard or even impossible to make small adaptations to the dialog or employ error recovery strategies. Nevertheless, being able to learn an optimal strategy or implicitly modeling the uncertainty can provide very useful when embedded as probabilistic sub-dialogs.

In the future we will further extend our language by describing more recent variances of the described patterns.

## Acknowledgements

The authors would like to thank our shepherd Johannes Koskinen for his valuable input. We would also like to thank the members of the writer's workshop at the VikingPLoP 2013 conference for their additional thoughts that helped us improving our work.

### References

[1] James O. Coplien. *A generative development-process pattern language*, pages 183–237. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[2] Melanie Hartmann and Daniel Schreiber. Prediction algorithms for user actions. In Ingo Brunkhorst, Daniel Krause, and Wassiou Sitou, editors, *15th Workshop on Adaptivity and User Modeling in Interactive Systems*, 2007.

[3] Gerard Meszaros and Jim Doble. Pattern languages of program design 3. chapter A pattern language for pattern writing, pages 529–574. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] J Nielsen. Classification of dialog techniques. *ACM SIGCHI Bulletin*, 1987.

[5] T. Reenskaug. Models - views - controllers. Technical report, Xerox Parc, 1979.

[6] Dirk Schnelle-Walka and Stefan Radomski. A pattern language for dialogue management. In *Proceeding of VikingPLoP 2012*, Apr 2012.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland