Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

03.04.2019

Lecture is every week Wednesday 09:00 - 11:00.

06.03.2019:	topic, teams
13.03.2019:	TISS registration, initial PR
20.03.2019:	other registrations, guest lecture
27.03.2019:	PR for first issue done, second started
03.04.2019:	first issue done, PR for second
10.04.2019:	mid-term submission of exercises
08.05.2019:	(HS?)
15.05.2019:	
22.05.2019:	
29.05.2019:	
05.06.2019:	final submission of exercises
12.06.2019:	
19.06.2019:	last corrections of exercises
26.06.2019:	exam

Popular Topics

14	tools
9	testability
9	code-generation
7	context-awareness
6	specification
6	misconfiguration
6	complexity reduction
5	validation
5	points in time
5	error messages
5	auto-detection
4	user interface
4	introspection
4	design
4	cascading
4	architecture of access
3	configuration sources
3	config-less systems
2	secure conf
2	architectural decisions
1	push vs. pull
1	infrastructure as code
1	full vs. partial
1	convention over conf
1	CI/CD
0	documentation

Pull Requests

- build server and reviews take time
- please also add test cases and documentation
- put label “ready to merge” after build server, reviews, . . . are satisfied

Deadlines

- gradual reduction of points for missed deadlines
- if nothing was done before mid-term, only 50 % is possible

Examples:

- If 7 PRs were done for homework but none of them was done before mid-term, you get 15 instead of 30 points.
- If 7 PRs were done for homework but 2 of them were delayed, you get 22 instead of 30 points.

Team Work

Clarifications needed:

- Who does what?
- either one more complex or two more simple applications
- one needs to write instructions and specification for the other

Tasks for today

(until 03.04.2019 23:59)

Task

Fix misconfigurations in private repo.

Task

Add clarifications and fix feedback about homework/teamwork.
Calculate complexity of your teamwork.

Task

First issue done, PR for second issue and write some text in at least one other issue (if 5 issues are not yet assigned to you).

Tasks for next week

(until 10.04.2019 23:59)
mid-term submission of exercises

Task

Submit a first version of both teamwork and homework.

Does not need to be complete, important is that you get started.

Task

Second PR done, PR for third issue created and write some text in at least one other issue (if 5 issues are not yet assigned to you).

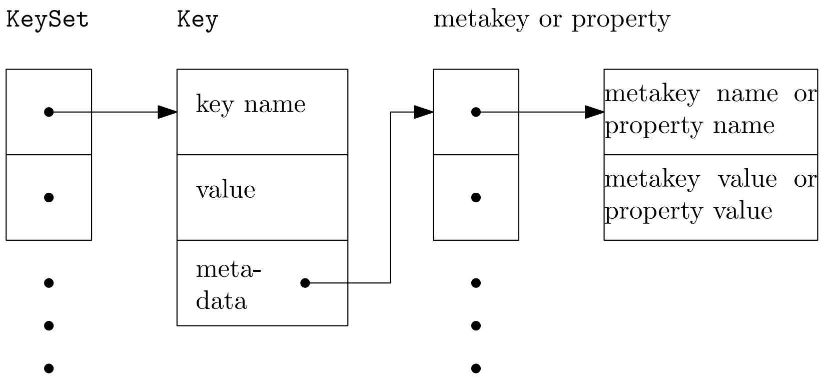
Task

Write one architectural decision for your teamwork or Elektra.

KeySet (Recapitulation)

Question

Describe the common data structure in Elektra.


KeySet
Key
metakey or property
key name
value
meta-data
metakey name or property name
metakey value or property value

Unnecessary Settings [6] (Recapitulation)

Question

How many settings are actually used?

- 6 % to 17 % of settings set by majority
- up to 54 % are seldom set
- up to 47 % of numeric settings have no more than five distinct values

Question

How can you reduce the complexity of configuration settings?

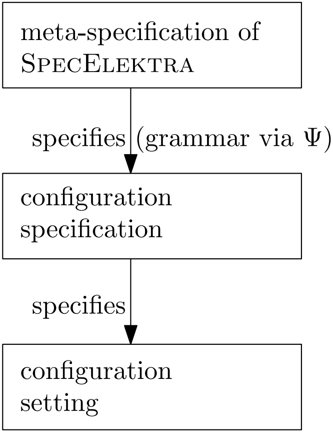
Answer

- Configuration Specification (restrictions, better design, . . .)
- unify formats, semantics, . . .
- avoid to have them (hard-code, better defaults, . . .)

Metalevels (Recapitulation)

Question

Describe the three Metalevels in Elektra.


meta-specification of SpecElektra
specifies (grammar via Ψ)
configuration specification
specifies
configuration setting

learning outcome:

- understand concepts of configuration specifications
- compare different ways to integrate configuration specifications
- remember templates to describe software architecture

Configuration Specification

1. Configuration Specification
   - How?
   - Example
   - Calculate Default Values
2. Elektrify
   - Definitions
   - Lightweight vs. Strong
3. Architectural Decisions

Task

Brainstorming: What can be part of a configuration specification? What can they be used for?

Q: "Configuration specification (e.g. XSD/JSON schemas) allows you to describe possible values and their meaning. Why do/would you specify configuration?"

- 58 % for *"looking up what the value does"*,
- 51 % it helps users to avoid common errors (*"so that users avoid common errors"*),
- 46 % to simplify maintenance,
- 40 % for rigorous validation,
- 39 % for documentation generation (for example, man pages, user guide),
- 30 % for external tools accessing configuration,
- 28 % for generating user interfaces,
- 25 % for code generation, and
- 24 % for specification of links between configuration settings.

Limitations of Schemata designed for Data

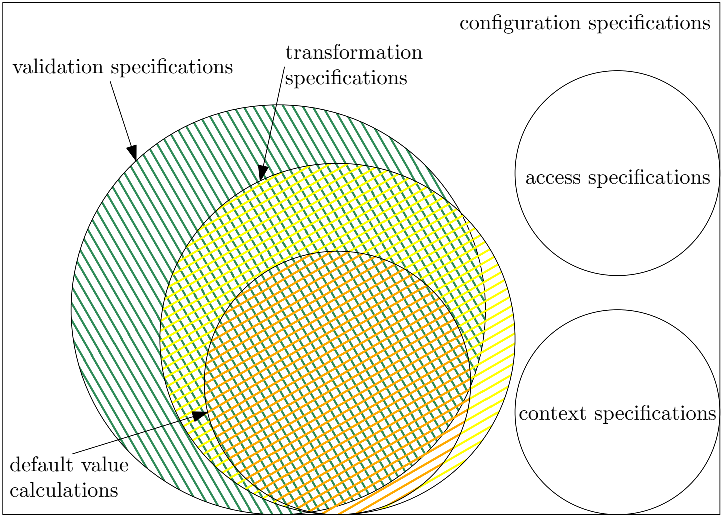
- like XSD/JSON schemas
- they are already very helpful but:
  - not key-value based
  - not easy to introspect
  - designed to validate data without semantics: file path vs. presence of file
  - not always possible to extend with plugins
  - tied to specific formats (e.g. XML/JSON)

Limitations of Zero-Configuration

- e.g. gpsd[1]
- broken hardware or protocols
- auto-detection may go wrong
- the configuration actually lives elsewhere (e.g., in the GPS devices)

[1] www.aosabook.org/en/gpsd.html

Types of Specifications


configuration specifications
validation specifications
transformation specifications
access specifications
context specifications
default value calculations

How?

Task

What do we mean with a configuration specification?

Task

Which requirements do we have for a configuration specification?

Requirements

- formal/informal?
- complete?
- should be extensible
- should be external to application
- open for introspection (for tooling)
- should talk to users
- should allow generation of artefacts

Grammar

$$\langle \textit{configuration specifications} \rangle ::= \{ \langle \textit{configuration specification} \rangle \}$$

$$\langle \textit{configuration specification} \rangle ::= \text{'['} \langle \textit{key} \rangle \text{']'} \langle \textit{properties} \rangle$$

$$\langle \textit{properties} \rangle ::= \{ \langle \textit{property} \rangle \}$$

$$\langle \textit{property} \rangle ::= \langle \textit{property name} \rangle \text{':='} [\langle \textit{property value} \rangle]$$

Example

```
[slapd/threads/listener]
default:=1
type:=long
```

Options

Environment and command-line options can be considered with:

```
[recursive]
  type:=boolean
  opt:=r
  opt/long:=recursive
  env:=RECURSIVE
  default:=0
```

Visibility

- idea: show only relevant settings for specific user group
- or disallow editing: accessibility
- requires user-feedback loops [6]
- most-used settings should be best visible (or even enforce them to be changed: against harmful defaults)
- think of your users (administrators),
  only expose what users need
- write an rationale why someone needs it
- visibility should not be an excuse to add not-needed settings

Example

```
[slapd/threads/listener]
visibility:=developer

[slapd/access/#]
visibility:=user
```

Task

Brainstorming: Now, how do we implement such a specification?

Possible Implementations

- tooling (GUI, Web UI)
- generate examples/documentation
- auto-completion/syntax highlighting/IDE support
- plugins in configuration framework (hide settings)

Task

Break.

Calculate Default Values

- idea: make default value better
- is the generalization of sharing configuration values
- can be combined with visibility
- can be derived from other configuration settings
- can be derived from context [4]
- can be derived from hardware/system (problem with dependences)
- XServer vs. gpsd

Examples

Sharing:

```
[slapd/threads/listener]
fallback/#0:=slapd/threads
```

Percentages
(e.g., configured image should be additionally cropped):

```
[image/width]
type:=long

[crop]
type:=long
check/range:=0-100
```

Examples

Context:

```
[slapd/threads/listener]
context:=/slapd/threads/%cpu%/listener
```

Calculation with conditionals plugin
(e.g., switch off GPS if battery low):

```
[gps/status]
assign:=(battery > ’low’) ? (’on’) : (’off’)
```

Question

How do we get such an specification now?

Answer

Elektrify: Make the application use a configuration library that has support for configuration specifications.

Elektrify

1. Configuration Specification
   - How?
   - Example
   - Calculate Default Values
2. Elektrify
   - Definitions
   - Lightweight vs. Strong
3. Architectural Decisions

Configuration Access APIs

An ***application programming interface (API)*** defines boundaries on source code level. Better APIs make the execution environment easier and more uniformly accessible.
Configuration access is the part of every software system concerned with fetching and storing configuration settings from and to the execution environment. There are many ways to access configuration [2, 3, 5]. ***Configuration access APIs*** are APIs that enable configuration access.

Configuration Access APIs

Task

Which configuration access APIs do you know?
What are the differences between these APIs?

For example:

- `char * getenv (const char * key)`
- `ConfigStatus xf86HandleConfigFile(Bool autoconfig)`
- `long pathconf (const char *path, int name)`
- `long sysconf (int name)`
- `size_t confstr (int name, char *buf, size_t len)`

Configuration Access Points

Within the source code the ***configuration access points*** are configuration access API invocations that return configuration values.

```
int main()
{
    getenv ("PATH");
}
```

Configuration Libraries

Configuration libraries provide implementations for a configuration access API.
Trends:

- flexibility to configure configuration access (e.g., `https://commons.apache.org/proper/commons-configuration/`)
- more type safety (e.g., `http://owner.aeonbits.org/`, code generation in next lecture)
- try to unify something (UCI, Augeas, Elektra)

Lightweight Integration

Specify already-existing configuration files:

```
[ntp]
  mountpoint:=ntp.conf
  infos/plugins:=ntp
```

Works well for configuration management tools.

Medium Integration

Having frontends that implement existing **APIs** decouple applications from each other. These applications continue to use their specific configuration accesses, but Elektra redirects their configuration accesses to the shared key database.
Possible APIs:

- `getenv` (implemented in `bindings/intercept/env`)
- open/close of configuration files

Also needs application-specific specifications.

Strong Integration

Change the application so that it directly uses Elektra.
Advantages:

- Elektra's features always available
- more type safety
- administrators can choose configuration file formats
- notification and logging
- only one parser involved
- no specification for binding needed
- no built-in defaults: everything is introspectable

Strong Integration

Different implementations strategies:

- have some application-specific API which uses `KeySet`
- use one of KeySet's language bindings
- use Elektra's high-level API (currently only C)
- code generation

Task

What will you use for the teamwork?

Architectural Decisions

1. Configuration Specification
   - How?
   - Example
   - Calculate Default Values
2. Elektrify
   - Definitions
   - Lightweight vs. Strong
3. Architectural Decisions

Software Architecture

- architecture is high-level description of the overall system
- use ready-made patterns and templates for architecture
- e.g., `http://arc42.org/`
- architectural decisions [1] essential (e.g., Chapter 9 in arc42)

Architectural Decisions

- describe decisions that lead to the architecture
- open decisions are high-level configuration
- useful to have patterns [1] and templates, too
- template: problem, constraints, assumptions, considered alternatives, decision, rationale, implications, related, notes

Why are configuration settings added?

The typical reasons are:

1. a requirement,
2. an architectural decision,
3. a technical need, and
4. an ad hoc decision.

in Configuration Specification

```
[slapd/threads/listener]
description:=adjust to use more threads
rationale:=needed for many-core systems
requirement:=1234
visibility:=developer
```

Conclusion

- alarming trend in number and complexity of configuration
- sharing, visibility and default value calculation may help
- but also more courageous decisions and periodical reevaluation
- both need abstraction: configuration specification

[1] Neil B Harrison, Paris Avgeriou, and Uwe Zdun. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4):38–45, 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.124.

[2] Dongpu Jin, Xiao Qu, Myra B. Cohen, and Brian Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 215–224, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2768-8. doi: 10.1145/2591062.2591191. URL `http://dx.doi.org/10.1145/2591062.2591191`.

[3] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 28–35. IEEE, May 2004. doi: 10.1109/ICAC.2004.1301344.

[4] Markus Raab and Gergö Barany. Introducing context awareness in unmodified, context-unaware software. In *Proceedings of the 12th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 218–225. INSTICC, ScitePress, 2017. ISBN 978-989-758-250-9. doi: 10.5220/0006326602180225.

[5] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 244–259. ACM, 2013.

[6] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs! Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 307–319, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786852. URL http://dx.doi.org/10.1145/2786805.2786852.