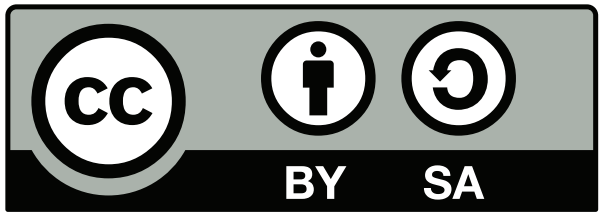
Configuration Management

Markus Raab

Institute of Information Systems Engineering, TU Wien

15.05.2018



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: different location: Complang Libary
15.05.2019:
22.05.2019: all 5 issues done
29.05.2019:
05.06.2019: final submission of exercises
12.06.2019:
19.06.2019: last corrections of exercises
26.06.2019: exam

Tasks for today

(until 15.05.2019 23:59)

Task

Fourth PR done, PR for fifth issue created.

Tasks for next week

(until 22.05.2019 23:59)

Task

All issues done.

Task

Continue teamwork and homework.

Popular Topics

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

Goals for today

learning outcome:

- evaluate a configuration system and decide about
  - use of code generation (recapitulation)
  - use of system-wide introspection
  - testability
  - time of validation

Introspection

Introspection (Recapitulation)

Question

What can introspection offer?

- unified get/set access to (meta*)-key/values
- access via applications, CLI, GUI, web-UI, ...
- access via any programming language (similar to file systems)
- GUI, web-UI can semantically interpret metadata

Internal Specification

For example, OWNER:

```
1 import org.aeonbits.owner.Config;
2
3 public interface ServerConfig extends Config {
4     int port();
5     String hostname();
6     @DefaultValue("42")
7     int maxThreads();
8 }
```

Question

Why do we need an external specification?

Introspection:

- needed as communication of producers and consumers of configuration
- the foundation for any advanced tooling like configuration management tools
- essential for ***no-futz computing*** Holland et al. [2]

External Specification

```
1 [port]
2 type := long
3 [hostname]
4 default := 42
5 [threads/max]
6 type := long
```

Advantages:

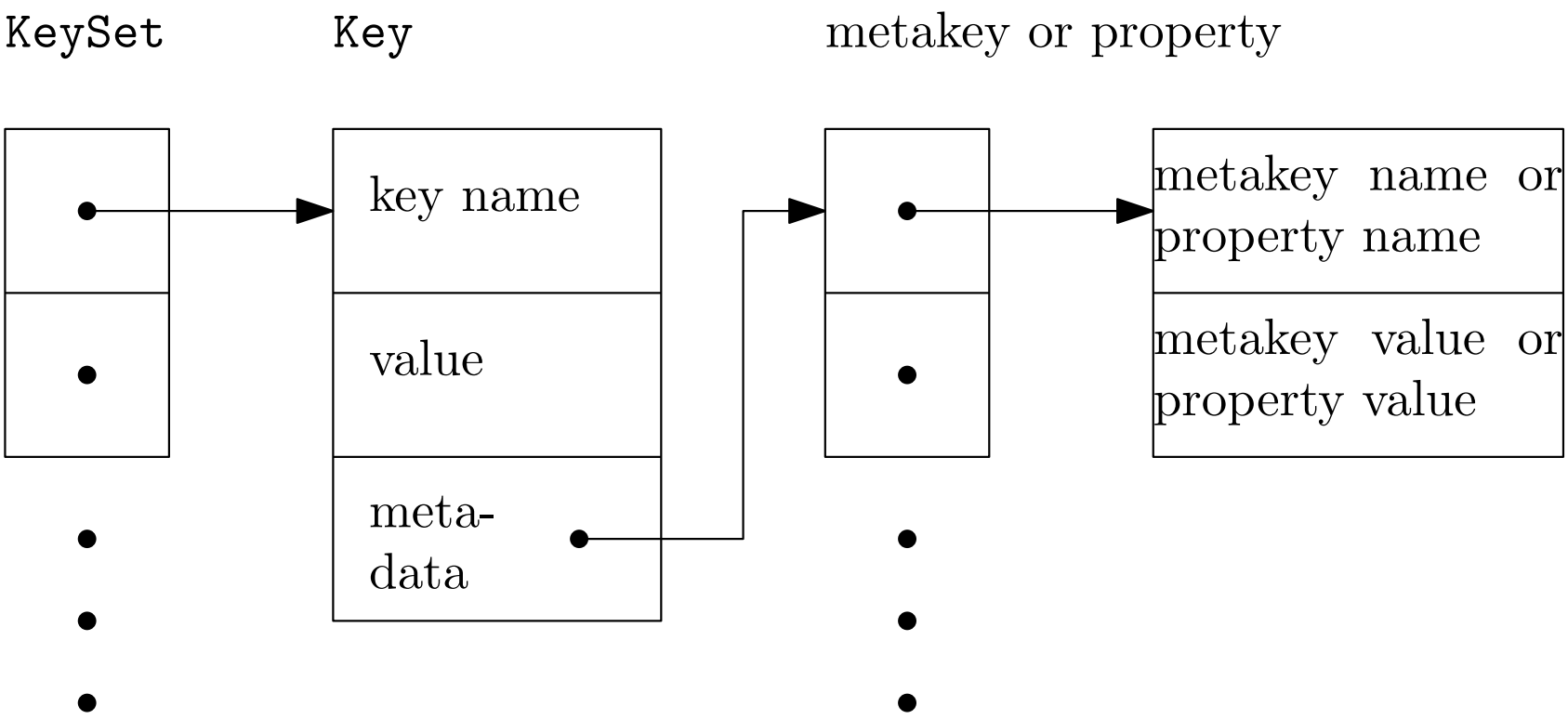
- are read and writable by other applications (introspection)
- we can generate the internal specification (code generation)
- we fulfill needs for configuration management tools

Other Artefacts (Recapitulation):

- examples (e.g., defaults)
- documentation
- auto-completion/syntax highlighting/IDE support
- tooling (GUI, Web UI)
- validation code
- parsing code (e.g., command-line parsing)
- configuration management tool code
- configuration access APIs

KeySet (Recapitulation)

The common data structure between plugins:



KeySet Generation (Recapitulation)

Question

Idea: What if the configuration file format grammar describes source code?

key `spec:/slapd/threads/listener`, with the configuration value `4` and the property default $\mapsto$ 1:

```
1 ksNew (keyNew ("spec:/slapd/threads/listener",
2                KEY_VALUE, "4",
3                KEY_META, "default", "1",
4                KEY_END),
5          KS_END);
```

Finding

We get source code representing the settings.

Possible Properties (Recapitulation)

For example, SpecElektra has following properties:

type represents the type to be used in the emitted source code.

opt is used for short command-line options to be copied to the namespace proc.

opt/long is used for long command-line options, which differ from short command-line options by supporting strings and not only characters.

readonly yields compilation errors when developers assign a value to a contextual value within the program.

default enables us to start the application even if the backend does not work.

(Recapitulation)

Question

Introspection vs. Code Generation?

- – more techniques for performance improvements with code generation
- \+ specification can be updated live on the system without recompilation
- \+ tooling has generic access to all specifications
- \+ new features the key database (e.g., better validation) are immediately available consistently

Implication

We generally prefer introspection, except for a very thin configuration access API.

Testability

Question

What do we want to test?

- That settings do what they should (devs and admins)
- That settings are properly validated (devs [7])
- Regression tests [5]

- Are all settings implemented?
- Are all settings used in tests?
- Are there unused settings in the code?

Matt Welsh from Google wrote in 2013:[1]

"Of course we have extensive testing infrastructure, but the 'hard' problems always come up when running in a real production environment, with real traffic and real resource constraints. Even integration tests and canarying are a joke compared to how complex production-scale systems are."

[1] What I wish systems researchers would work on. Retrieved from http://matt-welsh.blogspot.com/2013/05/what-i-wish-systems-researchers-would.html.

Jin et al. [3]

- Wants to improve configuration-aware testing and debugging
- Manual investigations for three applications
- Finds 1957 settings in Firefox ($2^{846} * 3^{1111}$) and 36322 in LibreOffice ($2^{4433} * 3^{31889}$)
- Finds unused settings: settings only in the source code
- Finds unsynchronized configuration settings

Requirement

Configuration setting traceability is a necessity.

Idea

Code generation helps to trace settings and to find unused settings.

Testing by developers:

- ConfErr [4] uses models of key board layout, psychology and linguistics. Tool injects possible misconfiguration.
- Spex [7] analyzes the source code to find misconfigurations. As by-product it extracts internal specifications (including transformation bugs).
- External specification can be directly used to generate test cases.
- Find unused configuration settings.

Task

Break.

Find Unused Settings

The first (optional) step of the algorithm is:

- Run all tests with code coverage.
- Check if generated code is executed.
- If it is, we know that the configuration setting is used in a test case. Otherwise, we know it is not tested by the test suite. All these untested configuration settings are remembered as candidates for the second step.

```
 1 KeySet findUnusedSettings (KeySet untestedSettings,
 2                            KDB kdb,
 3                            Builder build)
 4 {
 5     KeySet unusedSettings = {};
 6     KeySet configurationSpecification;
 7     kdb.get (configurationSpecification);
 8
 9     for (candidate: untestedSettings)
10     {
11         configurationSpecification.remove (candidate);
12         kdb.set (configurationSpecification);
13         build.recompile ();
14         if (build.wasSuccessful ())
15         {
16             unusedSettings.append (candidate);
17         }
18         configurationSpecification.append (candidate);
19     }
20
21     kdb.set (configurationSpecification);
22     return unusedSettings;
23 }
```

Early Detection

When are settings used?

Implementation-time configuration accesses are hard-coded settings in the source code repository. For example, architectural decisions [1] lead to implementation-time settings.

Compile-time configuration accesses are configuration accesses resolved by the build system while compiling the code.

Deployment-time configuration accesses are configuration accesses while the software is installed.

Load-time configuration accesses are configuration accesses during the start of applications.

Run-time configuration accesses are configuration accesses during execution not limited to the startup procedure.

Latent Misconfiguration

Phases when we can detect misconfigurations:

- Compilation stage in configuration management tool
- Writing configuration settings on nodes
- Starting applications (load-time)
- When configuration setting is actually used (run-time)

Problem

More context vs. easier to detect and fix.

As shown by Xu et al. [8]:

- 12 % – 39 % configuration settings are not used at all during initialization.
- Applications often have latent misconfigurations (14 % – 93 %)
- Latent misconfigurations are particular severe (75 % of high-severity misconfigurations)
- Latent misconfiguration needs longer to diagnose

Checkers as plugins

Using checkers as plugins exclude whole classes of errors such as:

- Invalid file paths using the plugin *"path"*.
- Invalid IP addresses or host names using the plugins *"network"* or *"ipaddr"*.

Because the checks occur before the resources are actually used, the checks are subject to race conditions.[1]
In some situations facilities of the operating system help[2], in others we have fundamental problems.[3]

---

[1] For example, a path that was present during the check, can have been removed when the application tries to access it.

[2] For example, we open the file during the check and pass `/proc/<pid>/fd/<fd>` to the application. This file cannot be unlinked, but unfortunately the file descriptor requires resources.

[3] For example, if the host we want to reach has gone offline after validation.

Example [8]

Squid uses `diskd_program` but not before requests are served. Latent misconfiguration caused 7h downtime and 48h diagnosis effort.

Finding

Configuration from all externals programs need to be checked, too.

Conclusion

- provide external specifications for other tooling and configuration management
- use code generation to keep internal specifications consistent with external specifications
- implement checkers as plugins
- execute checkers as early as possible, also for external programs executed later
- keep important resources allocated after checking

Preview

- Documentation
- Notification
- Context-Awareness

[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] David A. Holland, William Josephson, Kostas Magoutis, Margo I. Seltzer, Christopher A. Stein, and Ada Lim. Research issues in no-futz computing. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 106–110. IEEE, May 2001. doi: 10.1109/HOTOS.2001.990069.

[3] 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.

[4] Lorenzo Keller, Prasang Upadhyaya, and George Candea. Conferr: A tool for assessing resilience to human configuration errors. In *Dependable Systems and Networks With FTCS and DCC, 2008.*, pages 157–166. IEEE, 2008.

[5] Xiao Qu, Myra B. Cohen, and Gregg Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 75–86, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: 10.1145/1390630.1390641. URL http://doi.acm.org/10.1145/1390630.1390641.

[6] 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.

[7] 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.

[8] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, USA, November 2016.