Future-Proof Software-Systems: Summary

Key Points FPSS
Lecture: 06.01.2016
January 6, 2016:

- Legacy System Migration/Modernization
- Software Product Lines
- Domain Software Engineering

- Part 4A: Architecting for Resilience
  
  *General Resilience Architecture Principles*
A legacy system refers to outdated computer systems, eroded software architectures, old programming languages or badly supported applications software.
A very large part of systems in today’s use are legacy systems!

**bad:**
- very low agility
- weak resilience
- eroded architecture
- badly or not documented
- obsolete technology (HW & SW)
- large technical debt
- lost knowledge (people left)

**good:**
- invaluable *implicit* knowledge of the domain and the business processes
- stable operation (mature)
- good solutions/algorithms
- often: suprisingly good code
Future-Proof Software-Systems: Summary

Legacy systems must (at some time) be replaced or migrated

- High resistance to change for new business requirements (low agility)
- Weak resilience (attacks, faults, ...)
- Technology pressure
- New architecture paradigms
- Knowledge shortage

Legacy Software-System

Functionality & Data

How do we replace or migrate legacy systems?

⇒ By applying a migration strategy
<table>
<thead>
<tr>
<th>Type of Migration</th>
<th>Current State</th>
<th>Target State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replacing</td>
<td>Operational software. Cost, time and risk for a migration to high software has completely been rewritten, starting from the initial requirements</td>
<td></td>
</tr>
<tr>
<td>Re-Architecting</td>
<td>Operational software. Architecture paradigm has changed [e.g. monolithic architecture $\Rightarrow$ service-oriented architecture]</td>
<td>Software runs under the new architecture paradigm</td>
</tr>
<tr>
<td>Re-Engineering</td>
<td>Operational code running on an outdated execution platform or using an obsolete software technology</td>
<td>Code runs on the modern execution platform or uses modern software technology</td>
</tr>
<tr>
<td>Re-Factoring</td>
<td>Operational code, deficiencies in the program implementation</td>
<td>Improved code (quality criteria)</td>
</tr>
<tr>
<td>Reverse Engineering</td>
<td>Operational code, massive lack of documentation, of knowledge and of source code</td>
<td>System is sufficiently understood and documented to start migration</td>
</tr>
</tbody>
</table>
Our world runs on Billions of legacy lines of code which must be migrated/modernized in the next decades

⇒ Enormous challenge for the SW-community
Architecture Recommendations for Legacy System Modernization

1. Unambiguously specify the boundary of the system (Code & Data) to be migrated/modernized

2. Clearly assess the state of the legacy system (code, data, documentation, value)

3. Precisely define the migration/modernization goals (for code & data)

4. Choose a migration/modernization strategy based on risk, fit-for-future, cost & time and quality attributes (e.g. certification or validation etc.)

5. Select optimum tool support [Note: Many excellent tools available, search www]
Software Product Lines

A software product line is a set of software-intensive systems sharing a common, managed set of features, that satisfy the specific needs of a particular market or mission and that are developed from a common set of core assets

[Clements02]
Product Line Conception

Market

Company Strategy

Feature Model
Wave 1:
- X
- Y
Wave 2:
- Z
- R
...

Product Line Architecture

Wave 1
- Doc

Wave 2
- Doc

Wave 3
- Doc

Wave 4
- Doc
Economics of Product Line Development:

- **Total effort [€, t]**

- **Initial effort:**
  - Prod line def
  - Variability
  - ↑ Quality

- **Great advantage in cost, time-to-market and quality**

- **Single systems approach**

- **Product line approach**

- **Graph showing increasing total effort with increasing number of systems built.**

© Prof. Dr. Frank J. Furrer - WS 2015/16
Software Re-Use Power

Linda Northrop, SEI, 2008
Domain Software Engineering

Domain Software Engineering [DSE] =

an architectural *methodology*

for evolving a *software* system

that closely aligns to *business* domains
The Challenge: Divergence & Complexity

**Divergence =**
Mismatch between *Business Needs* and *IT-Implementation*

**Complexity =**
Property of an IT-system which makes it difficult to formulate its overall behaviour, even when given complete information about its parts and their relationships.
Future-Proof Software-Systems: Summary

Divergence

Business Units:
- Vocabulary
- Concepts
- Customs
- Knowledge
- Tradition
- ...

IT Units:
- Terminology
- Classes
- Programming
- Constraints
- Experience
- ...

Error-prone Transformation
Future-Proof Software-Systems: Summary

- Business Units
  - Business needs
  - Requirements

- IT Units
  - Specifications
  - Development
  - Integration
  - Deployment

Divergence

Misunderstandings, Loss of Precision
**Complexity**

**Essential** complexity

... is the *inherent* complexity of the system to be built

Essential complexity for a given problem *cannot* be reduced.

**Accidental** Complexity

... is *introduced* by our development activities or by constraints from our environment
Future-Proof Software-Systems: Summary

Business Units

- Business needs
- Requirements

IT Units

- Specifications
- Development
- Integration
- Deployment

Complexity

Translation: Accidental Complexity

Modeling

Accidental Complexity

Implementation

Accidental Complexity

Integration

Accidental Complexity

© Prof. Dr. Frank J. Furrer - WS 2015/16
We need to improve communication and understanding!

⇒ The promise and value of domain software engineering (DSE)
Which are the key elements of DSE (Domain Software Engineering?)

1. Understanding the Business/Application Domain in terms of the business
   (⇒ Domain Model)
2. Use of an ubiquitous language
   (Business ⇔ IT alignment)
3. Software: Implementation of Business Domain concepts
   (Concepts ⇒ Business objects ⇒ Programm objects)
The DSE concepts:

- **Business/Application Domain**
- **Bounded Context**
- **Domain Model**
- **Anticorruption Layer**

A **Domain** is a Sphere of Knowledge, Influence or Activity.

The **Bounded Context** is the Boundary of a Model.

A **Domain Model** is a representation of the Entities, Relationships and their Properties in a specific Application Domain.

An **Anti-Corruption Layer** is a method to isolate two domains or systems, allowing systems to be integrated without knowledge of each other.
Future-Proof Software-Systems: Summary

Ubiquitous Language (UL)

Domain Experts

Ubiquitous Language

Software Teams

Customer/Business

IT Organization

«Boxes & Lines»
Text

Boxes & Lines with semantics

UML, SysML

Ontologies

Formalization

low

high

© Prof. Dr. Frank J. Furrer - WS 2015/16
Conclusions:

- Domain Software Engineering (DSE) is a very promising software development methodology.
- It has the potential to massively reduce divergence and accidental complexity.
- It forces a close cooperation between business units (domain experts) and IT experts.
- The base for success is an accurate, current domain model in business terms (business concepts & objects).
- The software implementation must strongly reflect the elements of the domain model.
Architecture Recommendations for Domain Software Engineering (DSE)

1. Gracefully build up an Ubiquitous Language between Business/Customer and IT (Implementer)

2. Define a consistent and complete domain model (hierarchical because of the size)

3. Push the formalization as far as possible (without losing the business/customer)

4. Use the terminology from the domain model/ubiquitous language in the code

5. Keep the domain model and the code implementation strictly synchronized at all times
Part 4: Architecting for Resilience
Resilience is the *capability* of a system with specific characteristics before, during and after a *disruption* to absorb the disruption, *recover* to an acceptable level of performance, and *sustain* that level for an acceptable period of time.

- **Before** – Allows anticipation and corrective action to be considered
- **During** – How the system survives the impact of the disruption
- **After** – How the system recovers from the disruption

Continuous development of resilience leads to a sustainable system
(= path to future-proof SW-systems)
Resilience = Result of the relevant properties of the system

- Safety
- Security
- Integrity
- Availability
- Confidentiality
- Certifiability
- Reliability
- Diagnosability
- Standards adherence
- Accountability
- Traceability
- Non-Repudiation
- Fault-Tolerance
- Business Continuity
- Recoverability
- Graceful Degradation
- Auditability
- Performance
- Real-Time Capability
- Fail-Save Behaviour
- Survivability
- etc.
<table>
<thead>
<tr>
<th>#</th>
<th>System Quality Property</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0: irrelevant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10: highest importance</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Primary Characteristics**

1. Business Value | 10
2. Agility       | 10

**Resilience:**

3. Safety         | 9
4. Fault-Tolerance| 9
5. Compliance to laws & regulations | 9
6. Integrity (Sensor Data) | 9
7. Availability   | 8
8. Security       | 7
9. Diagnosability | 6

**Secondary Characteristics**

10. Resources (Memory, CPU, …) | 8
11. Compliance to industry-standards | 7
12. Usability (User Interfaces)   | 9

**Example:**

Automotive Domain

Resilience defined
all possible incidents

1. Specific Countermeasures (\(\leftrightarrow\) Risk Analysis)

2. Architectural Countermeasures (Principles)

3. Adaptive Behaviour («Autonomic Computing»)
Software Resilience

⇒ Architecture (Structure)
- governed by resilience architecture principles
Resilience Architecture Principles

<table>
<thead>
<tr>
<th>General Resilience Architecture Principles (Overarching Principles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safety Principles</td>
</tr>
<tr>
<td>Security Principles</td>
</tr>
<tr>
<td>Integrity Principles</td>
</tr>
<tr>
<td>Availability Principles</td>
</tr>
</tbody>
</table>

... etc.
Future-Proof Software-Systems: Summary

General (Overarching) Architecture Principles for Resilience

- R1: Fault Containment Regions
- R2: Single Points of Failure
- R3: Multiple Lines of Defense
- R4: Fail-Safe States
- R5: Graceful Degradation
- R6: Dependable Foundation (Infrastructure)
The consequences of a fault – the ensuing error – can propagate either by an erroneous message or by an erroneous output action of the faulty part.
R1: Fault Containment

Fault

System Part A
ERROR

Fault Containment Region

System Part B
ERROR

System Part C
ERROR

Build **error** propagation boundaries around each system part
Resilience Architecture Principle R1: 

**Fault Containment Regions**

1. Partition the system into fault containment regions 
2. Build error propagation boundaries around each system part (⇒ Interfaces) 
3. Provide sufficient redundant information about the intended behavior of the system parts (components)

**Justification:** A fault or incident causing an error or disruption in one part (component) of the system should not propagate to other parts of the system and thus cause a sequence of errors and failures.
Future-Proof Software-Systems: Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business</strong> Architecture (Business Processes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applications</strong> Architecture (Functionality)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Information</strong> (Data) Architecture (Information &amp; Data)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Integration</strong> Architecture (Cooperation Mechanisms)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Technical</strong> Architecture (Technical Infrastructure)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A single point of failure (SPOF) is a part of a system that, if it *fails*, will stop the *entire* system from working.
Example: Computer Network

Redundancy eliminates SPOF
Resilience Architecture Principle R2: **Single Points of Failure**

1. Identify possible single points of failure early in the architecture/design process (Note: single points of failure can occur on all levels of the architecture stack)

2. Eliminate single points of failure, e.g. by introducing redundancy or improving the architecture

**Justification:** Any single point of failure is a great risk for a resilient system. They must therefore consistently be avoided
R3: Multiple Lines of Defense

Threats
- Safety
- Security
- Availability
- ...

1st line of defense

2nd line of defense
R3: Multiple Lines of Defense

Resilience Architecture Principle R3:

**Multiple Lines of Defense**

1. For each threat and incident implement multiple lines of defense
2. For each line of defense use different methods, techniques and technology

**Justification**: If a line of defense is overcome as a consequence of an incident, the second (third, ...) line of defense may mitigate the impact of the incident
R4: Fail Safe States

*Fail-safe* means that a system will not endanger lives or property when it fails. It will go into a *fail-safe state* and stop working.

“As engineers we sometimes find designing systems to be well-built is much easier than designing it to fail predictably”

Peter Herena, 2011

[http://www.viewsender.com](http://www.viewsender.com)
R4: Fail Safe States

System represented as a state machine

start

Operational states

Operational transitions
Resilience Architecture Principle R4:

**Fail-Safe States**

1. Define and specify fail-safe states in the operation of the system (= states which do not endanger lives or property)

2. Define and implement trajectories to fail-safe states from all possible states in case of irrecoverable faults, errors or failures

3. The trajectories to fail-safe states must be automatic, short and dependable

**Justification:** When a system encounters an irrecoverable fault, error or failure a planned transition into a safe state will avoid damage
R5: Graceful Degradation

Graceful Degradation is with respect to a specific resilience property.

The graph shows the availability of a system with graceful degradation. The x-axis represents component failures, and the y-axis represents availability.

- **Availability**: The graph illustrates how the availability changes as component failures increase.
- **Tolerable Failures**: These are failures that do not impact the system's ability to operate within certain parameters.
- **Impacting Failures**: These failures have a significant impact on the system's performance.
- **System Failure**: This occurs when the system is unable to function safely.

The system can degrade gracefully to maintain some level of availability before reaching a state where it fails completely.
Future-Proof Software-Systems: Summary

R5: Graceful Degradation

Fault Tolerance: successful **redundancy**

**Graceful Degradation = Fault Tolerance Engineering**
R5: Graceful Degradation

Resilience Architecture Principle R5:

**Graceful Degradation**

1. Investigate the possibility for graceful degradation in your planned system (= Business task)

2. Architect and implement proven graceful degradation technologies (for specific resilience properties, such as availability, performance, safety, security, ...)

3. Compensate failures by redundancy

**Justification:** The value of many systems is significantly improved if after a failure of a component the system operates in a (planned) degraded mode instead of stopping service
R6: Dependable Foundation (Infrastructure)

Use a *resilience infrastructure* as part of a dependable foundation for resilient software-systems

**Resilience Infrastructure:**
Set of proven resilience technologies and services supporting the resilience properties (availability, security, performance, ...) of software systems

**Resilience Engineer Roles:**
- Security Engineer
- Safety Engineer
- Availability/Performance Engineer
- ... (*all required resilience properties*)
R6: Dependable Foundation (Infrastructure)

Dependable resilience mechanisms

1960

Application Software

Technical Infrastructure

1980

Application Software

Infrastructure Services

Technical Infrastructure

2000

Application Software

Infrastructure Services

Technical Infrastructure

Commodities Sourcing

2020

Application Software

Infrastructure Services

Technical Infrastructure

Commodities Sourcing

© Prof. Dr. Frank J. Furrer - WS 2015/16
Resilience Architecture Principle R6:

**Dependable Foundation (Infrastructure)**

1. Use a resilience infrastructure as part of a dependable foundation for resilient software-systems

2. Only use *proven* resilience technologies and services supporting the resilience properties (availability, security, performance, …)

3. Whenever possible use industry-standard based resilience techniques (But avoid vendor lock-in)

**Justification**: An implementation of proven resilience techniques in the form of industry-standard products forms a valuable, trustable resilience foundation
Exams Winter Term 2015/16

Planned dates (oral exams):
Wednesday, February 3, 15:00 – 18:00
Thursday, February 4, 09:00 – 11:30

Please write an email to katrin.heber@tu-dresden.de (Secretary of the Chair of Software Technology). She will schedule your timing of the exam.

Alternate date: If none of these dates suits you, there will be another exam time after the beginning of the Summer Term 2016. The date will be announced later. Please register with katrin.heber@tu-dresden.de
Future-Proof Software-Systems: Summary

Antrittsvorlesung Prof. Dr. Frank J. Furrer
21. Januaury 2016 / 13:00 – 14:15
http://st.inf.tu-dresden.de/fjfurrer/

13:00  BEGRÜSSUNG
   Dekan der Fakultät Informatik &
   Prof. Dr. Uwe Aßmann | Technische
   Universität Dresden

13:15  ANTRITTSVORLESUNG
   SOFTWARE
   – Gratwandernung zwischen
   Erfolgsgeschichten und Katastrophen?
   Prof. Dr. Frank J. Furrer | Technische
   Universität Dresden

14:15  KAFFEEPAUSE

15:00  Emergenz in Cyber-Physical System of
   Systems (CPSoS)
   Was führt zu emergentem Verhalten in Cyber-
   Physical System of Systems?
   Prof. Dr. Hermann Kopetz | Technische
   Universität Wien

15:45  Gotische Kathedralen und Software-
   Architekturen
   – Gibt es da irgendeinen Zusammenhang?
   Prof. Dr. Manfred Nagl | RWTH Aachen

16:30  Building correct cyberphysical systems
   – and how to improve current practice
   Prof. Dr. Susanne Graf | VERIMAG Grenoble

© Prof. Dr. Frank J. Furrer - WS 2015/16
January 20, 2016:

- **Part 4B**: Architecting for Resilience
  - *Specific Resilience Architecture Principles*
  - *Autonomic Computing*