FAILURES CAUSE LOSSES

- Accidents cause huge economic losses every year
- How can we avoid these costs?
Learning objectives

Become familiar with a Best Practice Framework for designing and implementing large and complex software-intensive Systems.

Learn how to apply the framework in large and complex projects.

GOALS

Our Challenge

Ways to create Good Architecture

Ways to create Bad Architecture

If you think good architecture is expensive, try bad architecture.

Joseph Yoder, Brian Foote
SURROUNDED BY FORCES

External and Internal Forces
(Push)

External and Internal Forces
(Pull)

FROM YESTERDAY’S FAILURES
to TOMORROW’S BEST PRACTICES

Architects need
Systematic Architecture Design
instead of Ad-hoc Design
Best Practices that guide their
activities in the whole lifecycle

A framework of best practices gives
guidance how to
create stable architectural design
assess and improve architecture
realize architecture
maintain and evolve architecture

Architects
Best
Practice
Framework
Skills, Competences, Experiences
Methods & Tools
Activities & Best Practices
ARCHITECTING

Source: wikipedia

SYSTEMATIC ARCHITECTURE DESIGN PROCESS
WHERE WE ARE:

- Architecture Enforcement
- Systematic Re-use
- Commonality/ Variability
- Qualities
- Architecting Baseline
- Architecting Process
- Scopes & Boundaries

TWIN PEAKS (PROBLEM)

If requirements are fuzzy, architecture design can’t be stable
If customers or product owners do not know what they need, good architecture is impossible
⇒ Requirements Engineering and Architecting can’t happen sequentially
What is the alternative?
TWIN PEAKS (SOLUTION)

In high uncertainty*, pure sequential processes do not work, because
Requirements engineers do not know if their specification is feasible
Architects do not know whether they can meet the specification

**Twin Peaks** is an explorative approach:
- Requirements engineering, competence ramp-up, and architecting happen in parallel and with tight interaction
- Feasibility prototypes to clarify and discuss open issues early

Twin Peaks helps mitigate risks and reduce waste

*such as unknown domain, insufficient technology knowledge, unavailability of a reference architecture

---

UNDEFINED SCOPE AND BOUNDARIES (PROBLEM)

Without clear scoping and definition of boundaries, it is undefined what a product should include or exclude, e.g.,
- Which functionality should be implemented in product/solution?
- What is provided externally?
- How do both parts interact?
- What is the domain and what are the subdomains?
- What is common and what is variable?

As a consequence, architects cannot focus on essence.
COMMON SCOPING PITFALLS

Pitfalls of Scoping

Wrong scope: system does not provide expected value
Too broad scope: development ends up with extra functionality no one needs => more value and more complexity than needed
Too small scope: system does only provide a subset of required functionality => less value than needed

DEFINED SCOPE AND BOUNDARIES (SOLUTION)

Define clear scope: what is internal and what is external (incl. interfaces, actors)

Tools for Scoping:
- Use case diagrams
- Business Drivers
- SWOT-Analysis
- C/V Analysis
- Domain-Driven Design
- Feature Modeling
- Model-driven Development
- Context Diagrams
- Architecture Styles & Patterns

Leverage a problem domain language to express scope & boundaries

Derive interfaces, formats, and protocols used at the boundaries

Use Cases show the scope of the system in terms of functionality

Context Diagrams depict the boundary
The term **domain model** is overloaded

In software architecture design a domain model is a conceptual and idiomatic model that incorporates **Components, Behavior & Interaction, Data**

It is more than a glossary as it comprises dependencies of concepts as well

- **Problem domain models** capture the core entities of the problem domain
- **Domain Driven Design** structures domains in subdomains
- **UML class diagrams** are not always the best option to define a domain model
- **DSLs** help formalize mature or simple (sub)domains

**DEFINING A (PROBLEM) DOMAIN MODEL**

**Input:**
- Use Cases, User Stories

**Output:**
- Domain Objects such as Warehouse Topology, Order Management
  - Static and Dynamic Relations between Domain Objects such as call dependencies (Execute Transportation Order) and classifications (Key Customer, Customer)
  - Information passed or maintained such as Purchase Order

**A Domain Model needs buy-in of Stakeholders**
WHERE WE ARE:

PREREQUISITES FOR SYSTEMATIC ARCHITECTURING

Understanding of Problem Domain
High Quality Requirement specification
  at least 30% of all requirements
  almost all high priority requirements
Knowledge of Business Drivers, Goals, Strategy,
Knowledge of Constraints & Risks
Risk mitigation in Place
Agile/Lean Process Model
Technology & Tool Competence in Solution Domain
Adequate Staffing
DERIVING ARCHITECTURALLY-SIGNIFICANT REQUIREMENTS

Specification of Architecturally-Significant Requirements
- Functional Requirements
- Constraints & Infrastructures

Quality Attributes
- Operational Qualities: user experience such as performance, security, reliability, ...
- Developmental Qualities: developer experience such as maintainability, modifiability, ...

SYSTEMATIC ARCHITECTURAL DESIGN FROM 30.000 FEET

Get (subset of) requirements, Business Goals and Constraints

Analyze and give feedback until high quality achieved

Prioritize and derive architecture-significant requirements

Functional design

Quality attribute driven design: Operational and Developmental

Evolution of Architecture Baseline

Executable Walking Skeleton

Beta Test (optional)

Repair and Improvement

Quality Check by Reviews and Tests

Driver: Even in agile and lean processes we need some front loading with respect to architecture design

Mind the Gap! It depends on the context whether a requirement is in focus of strategic or tactical design
**ARCHITECTURE BASELINING**

The Architecture Baseline
- Captures the essence of a system and represents the initial walking skeleton
- Contains a subset of the whole architecture and is derived from the architecturally most significant drivers
- Is restricted regarding its number of abstraction layers
- Must be sufficiently stable and flexible to guide further refinement and evolution

Note: It might be a challenge to balance stability and flexibility

Mind contradicting definitions of “architecting baseline” in literature

Architecture baseline is the prerequisite for refinement and evolution

Source: wikipedia

**WALKING SKELETON METAPHOR**

Skeleton initially represents architecture baseline and is extended slice by slice with functionalities or qualities:
- **User Scenario**, e.g., adding all muscles, nerves, ..., needed to move a finger
- **Quality Attribute Scenario**, e.g., response time of finger movement

After integration of new slices, Regression-Testing required which is known as **piecemeal growth**
ARCHITECTURE DESIGN IS SYSTEMIC DESIGN (I)

Strategic Design

- Architectsing = Systemic Design
- Strategic Design = Strategic Design + Selection of Global Infrastructures for Tactical Design

Tactical Design

ARCHITECTURE DESIGN IS SYSTEMIC DESIGN (II)

Strategic Design

- Consideration of important requirements with systemic effect but only up to a maximum abstraction level
- Infrastructure for Tactical Design - Intersection of Tactical and Strategic Design

Tactical Design

- Everything else

System design is a sequence of design decisions. Architecture comprises all decisions with systemic impact:
- Strategic Design is architectural design
- Tactical infrastructures with systemic impact are architectural design as well
 LIMITING ABSTRACTION DEPTH

Limit Abstraction Depth
Rule of Thumb: use around 3 or 4 levels
System
Subsystems
Components
Semantics of terms like Subsystem or Component need to be defined by development organization!

PRECEDENCE RULES

Qualities shouldn’t be injected accidentally. Their concrete integration is driven by priorities:
Functional design before Quality Attributes
Operational Quality before Developmental Quality
E.g., Extensibility precondition: we must know what to extend
Strategic Design before Tactical Design
High Priority Requirements before Low Priority Requirements
THE STRATEGIC DESIGN TRIANGLE (SOLUTION)

**Three main ingredients of strategic design**
- User scenarios
- Quality-Attribute Scenarios
- Core Infrastructures for Distribution, Concurrency

**Core Infrastructures**
- Infrastructural requirements of functional core are mapped to Core Infrastructures
- Quality attributes cause integration of additional Infrastructures
- Core needs to consider hardware constraints and is addressed before quality attributes
- Low Level Layer of wrapper façades to ensure technology independence

POSTPONING DECISIONS (PROBLEM)

Software development is subject to frequent changes and uncertainty
Software architects may postpone decisions as they don’t want to be responsible for
- wrong decisions
- risks introduced by uncertainty

But: If decisions are not made, the project will be delayed
GOOD DECISION MAKING (SOLUTION)

Software architects need courage to make decisions. They are aware of risks and their mitigation can explain the rationale of their decisions. They are open to obtain and give feedback. They do not try to make everyone happy. They integrate management in decision making. They trace all consequences of such decisions (requirement traceability, architecture tracking!). They rollback early from wrong decisions.

Wrong decisions can be more beneficial than lack of decisions.

WRITE-ONLY DOCUMENTS (PROBLEM)

Documenting appears to be a nasty duty mostly done during idle phases or after project completion without consideration of the target audience.

Documentation No-GOs:
- Outdated or wrong information
- Explanation of What but not of Why
- Lack of important information or inclusion of unimportant facts
- Lack of coordinating main author
Documents are core assets subject to versioning, change management, and reviews contain the What, Why and How with ...

- one responsible main author
- clear and comprehensive vertical and horizontal structure
- Considering the needs of the target audience focus on readability and understandability
- good habitability of information provided (KiSS)
- regular but not too frequent updates (e.g., after each increment)
- usage of domain language(s)
- balance between repetition and references

Good Content
- Enlightenment guaranteed

MEMENTO (PROBLEM)

- Symptom: deceleration of project progress by repeating discussion of already clarified issues
- After a few weeks colleagues won’t remember the outcome and restart discussions
- You feel captured in a time loop which decreases productivity without yourself even noticing it
PROJECT DIARY (SOLUTION)

Introduce a project diary* for everything important and not documented elsewhere
- Decisions and their rationale
- Discussions and their results
- Minutes

Helps remember decisions and discussions
With a diary you can
- Prevent unproductive and repeating discussions
- Understand the whole project much better
- Track project history and decisions

Source: Basebuilders.com

* You may use a Wiki for this purpose

OVER THE FENCE (PROBLEM)

Sometimes, specifications are thrown over the fence from one role to the other(s)
- Requirement specification to architects
- Architecture specification to developers

The message to recipients is
- I do not care, anymore. Now, it is your problem! (see S.E.P.)
- You may have questions, but I am not available
- The document is so comprehensive that only morons are not capable of understanding its brilliance

Source: Business Insider.com
Authors of a document are responsible for its usability. They share responsibility for all consequences that result from the document. Documents are never thrown over the fence but introduced, motivated, explained, and handed over to recipients. Authors need to be available in the succeeding phases to answer questions of recipients, obtain feedback and use it to improve their document. But communicate in an efficient, effective and motivating way.

Without documented mandatory guidelines, rules, coding conventions..., code bases erode over time, since each developer uses her own interpretations, styles and idioms. Other extreme: ambiguous, contradicting Guidelines. Dissatisfaction guaranteed.
GUIDANCE FOR DEVELOPERS (SOLUTION)

Strengthen implementation quality, understandability & developer acceptance with a set of policies that

- are mandatory
- are enforced
- are not contradicting
- make sense
- help avoid architecture and code issues
- address all stakeholders and technologies
- do not omit important aspects
- are acceptable for "code users"
- are documented
- are easy to understand and remember

WHERE WE ARE:

- Architecture Enforcement
- Systematic Re-use
- Commonality/ Variability
- Architecture Baseline
- Architecting Process
- Scopes & Boundaries

Source: J. John (British pastor)
WAR STORY – LACK OF BASELINE

Analyze this real world example
Except for the names it contains the original architecture baseline design
Would you be able to build a system using this “architecture”? If not, why not?

CREATE AN ARCHITECTURE BASELINE

According to agile methodology assets are built iteratively and incrementally
Architecture is a core asset itself and the backbone of the system
**Consequence:** We must design an architecture baseline for building and evolving systems
Consider the Baseline as the bones of the walking skeleton
Incremental design with intermediate results that are executable applications with product quality systems (walking skeleton) is our safety net. In each iteration the number of slices, breadth, stability grow. For systematic design high risk respectively priority is addressed first.

**INGREDIENTS FOR A BASELINE**

- Fundamental Structure, i.e., core, main responsibilities, relationships, interfaces, and collaborations
- Infrastructures for distribution, concurrency, etc.
- Approach for addressing external quality, i.e., mission critical, strategic quality attributes
- Guiding principles and Directives for architecture design
- Traceability between architecturally relevant requirements and architecture decisions

Hardware systems have baselines as well.

IBM architecture baseline for event processing.
**Separation of Concerns**

Strictly separate:
- Business functionality,
- Infrastructure functionality,
- Support and management functionality.

Otherwise, maintainability will be reduced.

Separation of concerns is the precondition for reducing complexity and increasing understandability & modifiability.

**Strategic Design**

Strategic design helps focus on essence:
- Fundamental approach for baseline architecture creation
- Addresses commonality plus architectural infrastructure for variability

Without strategic design thinking you do not have a safety net.
WHERE WE ARE:

- Architecture Enforcement
- Systematic Re-use
- Commonality/ Variability
- Qualities
- Architecting Baseline
- Architecting Process
- Scopes & Boundaries

CLASSIFICATION OF QUALITIES

Internal: implicit properties that lead to low expressiveness, ...
External: explicit properties observable by users of product or code
Operational: behavior that is externally observable (end user usability)
Developmental: usability of design and code base (i.e., habitability)

Quality Attributes: we do not use the term "nonfunctional requirements" (requirements that do not work) but "quality attributes".

Internal Quality

External Quality

Operational Quality Attributes
- Performance
- Security
- Safety
- Availability

Developmental Quality Attributes
- Maintainability
- Reliability
- Extensibility
- Interoperability
INTERNAL AND EXTERNAL QUALITY

**External Quality** addresses qualities related to the behavior of the system.

**Internal Quality** addresses qualities related to the structure of the system.

LACK OF EXTERNAL QUALITY CAN CAUSE DISASTERS

- **Tacoma Narrows Bridge** (1940) collapsed due to neglecting resonance forces.
- **Radiation Therapy System Therac-25**: lack of quality and software failure caused 3 deaths.
- **Challenger Disaster**: caused by unsafe rocket booster design.
- **Ariane 5 explosion**: due to overflow exception.
- **Titanic** crashed: "Even God himself could not sink this ship!"
**HANDLING QUALITY ATTRIBUTES**

1. Define quality attribute scenarios
2. Aggregate and prioritize quality attribute scenarios
3. Apply design tactics for quality attribute scenarios
4. Implement

**TRADEOFF AND SENSITIVITY POINTS**

Two aspects need consideration:
- A point in the architecture that affects a particular quality attribute is a **Sensitivity Point**.
- A point in the architecture that is a sensitivity point for more than one quality attribute is a **Tradeoff Point**.

Tradeoff Points cause risks. They must be handled with care as they may be sensitivity points for “contradicting” quality attributes that negatively affect each other, e.g.: performance & modifiability.

**A WAR STORY**

Name: Tradeoff by Surprise
Context: Highly Scalable System for Prepaid Phone ticketing
Problem: A strict Layers Architecture was introduced to allow the flexibility of modifying system parts. Nonetheless, every time the database scheme in the lowest layer changed, the client apps in the highest layer crashed.
Consequences: It turned out that client app developers did ignore the layered system and used direct access to the low level layers.
QUALITY ATTRIBUTE CONSTRAINTS

Quality attributes can’t be combined in arbitrary ways.

Example: According to the CAP theorem you can only have 2 of 3:
- **Consistency**
- **Availability**
- **Partition Tolerance**

**Consistency**: you always see the correct information but it may take some time.

**Availability**: you always get information which may not be current.

**Partition Tolerance**: the system is able to cope with partial failure.

AD-HOC INTEGRATION OF CORE INFRASTRUCTURES (PROBLEM)

**Wrong Assumption**
- For infrastructures such as concurrency a one-size-fits-all approach will do.
- Usage of infrastructure is independent of problem context.
- However, that does not work:
- Infrastructures support different usage patterns.
- Architectures are tightly coupled with infrastructures.

**Implication**: It depends on problem and scenarios which infrastructure to prefer and how to use and deploy it.

Multi Threading must be applied appropriately, otherwise a Big Ball of Threads is lurking in the design.
INFRASTRUCTURES (SOLUTION)

Infrastructure use depends on problem at hand, e.g.,
- In image processing a Concurrent Pipes & Filters architecture is useful
- For control systems Half Sync / Half Async is the better choice
- For automation/robotics a Concurrent Reactor model might be the way to go

Analyze the problem, check systems with similar characteristics, and then consider which architecture respectively infrastructure to use and how.

BUTTERFLY EFFECT (PROBLEM)

The more tradeoff points in an architecture, the more risks.

Why? One small change may cause multiple unwanted effects in various places.

This holds for all architectural hotspots (design centers).

In Product Line Engineering and Ecosystems risks/debt have multiple impact.

An issue in one of the core assets affects all applications.

We must keep this under control.
REDUCING RISKS OF BUTTERFLY EFFECTS (SOLUTION)

Ensure Traceability and use Architecture Tracking
Only create tradeoff points where inevitable
Make tradeoff points explicit
  a responsibility of the whole development team
Treat them as risks: identify, assess, mitigate, monitor them
Use feasibility prototypes to avoid Butterfly Effects

WHERE WE ARE:

Architecture Enforcement
Systematic Re-use
Commonality/ Variability
Qualities
Architecture Baseline
Architecting Process
Scopes & Boundaries
OPEN DO OR (PROBLEM)

Pitfalls when dealing with Commonality & Variability:
- addressing both at once
- opening the architecture for variability w/o closing backdoors
- emphasizing variability instead of commonality
- using undocumented interfaces - it is like breaking into a house

OPEN/CLOSE(D) (SOLUTION)

Commonality addressed first => Invariant parts become design centers
Variability points introduced by opening the architecture
Protect other system parts from unwanted access

Step 1: Design Commonality
Step 2: Open for Variability
Step 3: Only open variability point but close rest (sandbox)
VARIABILITY MANAGEMENT
(SOLUTION)

Variability has a business interface covered by

- Domain modeling: defining domains, subdomains, scope, contexts => Product Line
- Commonality and Variability

- Domain potential analysis: future changes in features, technology => future variation points

- Commonality/Variability model: e.g., a Feature model => current Variation points

Note: these artifacts are not static but evolve over time due to product & technology changes

Architects define explicit variation points according to variability requirements

VARIABILITY IN SPACE/TIME

Variability is multidimensional:

- Variability in Space (current variation points)
- Variability in Time (future variation points)

Do only consider known and expected variability!
HANDLING VARIABILITY AS AN AD-HOC ACTIVITY (PROBLEM)

Considerations:
Variability must be defined, planned, designed and implemented
Binding Time, Variability, Architecture & Design are not mere technical aspects
Architects & Developers are in charge of managing Variability
Variation points should always be explicit

Source: nasa.gov
Many forces have an impact on variability which also holds for non software systems

VARIABILITY – WHO BINDS THE VARIATION POINT

Architects
Developers & DevOps
Administrators
Users
Technical Service Staff
Integrator
Another System
System itself
ADDITIONAL DIMENSIONS OF VARIABILITY

**Granularity of Variation**
- Subsystem
- Layer
- Component
- Class
- Service
- Algorithm
- Driver
- Library

**Kind of Variation**
- Extension
- Replacement
- Removal
- Reconfiguration

**Implementation of Variation**
- Structure: e.g., topology
- Behavior: e.g., Algorithm
- Control Flow: e.g., Rules, Workflows
- Environment: Operating System, Middleware, UI, Persistence Providers, Hardware

**TARGET OF VARIATION**

**Variation in Function**: e.g., optional .NET adapter

**Variation in Technology**: e.g., upgrading from Windows 7 to Windows 10

**Variation in Environment**: e.g., integration in environment - from Java to REST-based APIs

**Variation in Data**: e.g., adaptation of message formats

**Variation in Quality Goals**: e.g., high performance LAN for connection with server

**Variation in Control Flow**: workflow and rule changes

Product Solution
VARIATION DEPENDENCIES & CONSTRAINTS

Variation points are no disconnected islands but may depend on each other be constrained.
Without constraints the variation would be too complex to handle.
Example for a Constraint: customer may choose a high end engine exclusively for sports cars.

Binding Type:

- **Optional:** e.g., advanced search can be added to online shop
- **One-of-many:** e.g., selecting kind of payment
- **Multiple-of-many:** e.g., favorite departments in store

TYPES OF VARIATION
**Positive Variability**
Minimum core; features are added (high degree of variation)

**Negative Variability**
Maximum core; features are removed (low degree of variation)

**Structural Variation**
i.e., the system is generated from a model (⇒ snow crystal)

**Non-Structural Variation**
originates from feature trees, natural language, ...

**Structural and Non-Structural Variations**
Variability in the problem domain must be implemented in the solution domain using mechanisms such as:
- Language-features Inheritance, Overloading, Interfaces, Traits
- Runtime Compilation
- Preprocessing
- Component Reuse
- Change of Binary Artifact
- Shared Libraries, Class Loaders
- Code variation (Preprocessors, Implicit DSLs, Generators)
- Configuration
- Adaptation and Composition such as service composition
- Reflection with Meta Objects

FROM PROBLEM TO SOLUTION DOMAIN

Binding time
- Modeling time
- Design time
- Implementation time
- Compile time
- Build time

Link time
- Deployment time
- Load time
- Run time
- Maintenance time

VARIABILITY & PATTERNS

Patterns that support variability
- Strategy
- Decorator
- Visitor
- Observer
- Plug-in Architecture
- Component Configurator
- Pipes & Filters
- Microkernel
- Abstract Factory
- ...

Source: stanford.edu
Using Software Design Patterns in Information Visualization helps deal with variability
HOW TO DESCRIBE VARIABILITY

Feature Models
Meta Models
DSLs (e.g., Workflows)
Wizards
Configuration/Deployment Files
Natural Language

WHERE WE ARE:

Architecture Baseline
Architecting Process
Qualities
Commonality/Variability
Systematic Re-use
Architecture Enforcement
Scopes & Boundaries
Unsystematic or Ad-hoc re-use is a good recipe for failure:
- source code reuse without understanding it
- 90% component reuse problem - only 10% need to be adapted but this incurs high costs
- too small-grained reuse
- too coarse-grained re-use
- emphasis on binary reuse
- neglecting business drivers
- tactical instead of strategic reuse

Strategic reuse in the large is systematic re-use:
- Economics-driven
- Reuse of interrelated artifacts (code, test plans, documents, configurations, models)
- Approaches: Product Families, Ecosystems, Platforms
- Defined reuse process
  - Domain Engineering separated from Application Engineering
  - C/V Analysis for Core Asset Identification
  - Feature Models (Customer viewpoints)

Systematic (planned) reuse works
Ignoring Reference Architecture (Problem)

Without Reference Architecture, system development is complex.
Using a “wrong” Reference Architecture is even worse.
Creating a Reference Architecture is error-prone and tedious.
It needs a lot of efforts.
It violates the DRY principle.
It is inevitable if there is none.

Reference Architecture (Solution)

A Reference Architecture (RA) captures the essence of a domain.
It provides a domain vocabulary.
It comprises common entities and relations (i.e., the abstractions).
It addresses variability.
It includes guidelines for its instantiation.
It can be derived from a set of existing solutions, or from a domain model.
AD HOC REUSE IN THE SMALL (PROBLEM)

Reuse in the small has a couple of success stories:
- Patterns
- Design Tactics
- Best Practices

but many liabilities:
- insufficient RoI
- unmanageable and unmaintainable reusable entities
- too complex to handle for sets of interdependent components

OVERDOSE OF PATTERNS (PROBLEM)

Patterns comprise human intelligence and experiences.
They provide proven solutions for recurring problems in given contexts.
Using a software pattern makes sense if it complies with the needs.
Using many patterns is beneficial when applied carefully & correctly.
Treating patterns as building blocks is wrong.
Wrong pattern application may even increase design erosion.
SYSTEMATIC AND CLEAN PATTERNS APPLICATION (I) (SOLUTION)

Use patterns where possible and useful (second part is often ignored)
Mind the context and the forces as well as the consequences of applying a pattern
Statements like “I used all patterns of the Gang-of-Four book” indicate bad design
Patterns are no building blocks but blueprints. You can’t build a patterns library
Patterns may be implemented in infinite ways
Home-grown solutions may be the better choice sometimes. Example: Observer in Real-time Systems

SYSTEMATIC AND CLEAN PATTERNS APPLICATION (II) (SOLUTION)

Right pattern for the right purpose and scale: architecture patterns, design patterns, idioms, ...

Role-based Design:
Whenever pattern is applicable, identify roles and assign them to existing components. Only if needed, add new components. I.e., Patterns are integrated into the design
Note: one component may contribute to multiple patterns
**DO REUSE IN THE SMALL BUT IN A COMMON CONTEXT (SOLUTION)**

**Reuse in the Small**, a prerequisite for **Reuse in the Large**

Take core assets as example needed for complex, reusable functionality that is subject to the DRY principle.

Works best for reusable entities within same ecosystem (e.g., .NET Framework, syngo.via, ...)

---

**RE-USABILITY (PROBLEM)**

**DON'T reuse low quality components**  
they contain defective code  
... which may affect the whole system

Cost for reusing low quality components is mostly higher than the costs for creating them from scratch

Only reuse of high quality components increases sustainability
QUALITY REUSE (SOLUTION)

Check quality of reusable assets

**An unusable component is not re-usable**

Estimate costs for refactoring – i.e., is component expensive to reuse

Do not constrain analysis to the component but also investigate the enclosing context

Integrating a low quality component adds accidental complexity and debt

WHERE WE ARE:

- Architecture Enforcement
- Systematic Re-use
- Commonality/ Variability
- Qualities
- Architecture Baseline
- Architecting Process
- Scopes & Boundaries

Source: dailymail.co.uk
Reuse is a matter of scale
Architecture Drift is the gap between implementation and (conceptual) architecture.
Can often not be undone (due to high costs and time efforts).
Developers don’t cause drift intentionally but accidentally due to:
  Lack of information
  Wrong interpretation of information

Lack of Enforcement (Problem)

House (as planned)

UML?! Never again!

House (as built)
**LACK OF ENFORCEMENT (SOLUTION)**

Architects are responsible to guide & support implementation to visit all teams regularly to clarify open issues to obtain and use architectural feedback for improvement to coach and mentor key developers & architects

**A WAR STORY**

Name: Confusing Architecture Context: Telecommunications Problem: In a Telecommunications project a telephone conference service was designed. In the design there were two entities called “Conference Organizer” and “Conference Manager”. The developer made a (wrong) assumption. Consequences: Implementation had to be refactored

**KILL THE MESSENGER (PROBLEM)**

If developers have issues with the architecture it’s not due to inability to understand your ingenious design!

With a “It can’t be a problem in the architectural design”- attitude architects miss a lot of improvement potential

⇒ They should proactively and regularly ask developers and other stakeholders for feedback

It’s not a quality proof but a problem, when nobody understands the design!
CELEBRATE THE MESSENGER
(SOLUTION)

Consider feedback from developers and architects as a safety net
Profit from their combined experiences & skills
Merge theory & practice
Note: an “excellent conceptual idea” might have unwanted side effects
Consider feedback as a gift by reviewers who spend their time to provide useful information
⇒ A good feedback culture is important

SUMMARY OF SYSTEMATIC ARCHITECTING

The design process has two phases:
Phase 1 – establishing baseline
Phase 2 – actual implementation

Modus Operandi:
1. Start with functional design (higher priority use cases before lower priority use cases). Use happy day scenarios first before continuing with rainy day scenarios
2. Integrate core infrastructures for distribution, concurrency, deployment, battery saving, ...
3. Design (a) infrastructure for operational qualities (higher priorities first)
4. Design infrastructure for developmental qualities (higher priorities first) and tactical implementation
5. Assess, repair, and iterate to build the system increment by increment
Although software has no mass, it does have weight, weight that can ossify any system by creating inertia to change and introducing crushing complexity

-- Grady Booch