goto; conference aarhus

Seven Secrets

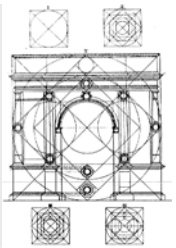
Every Architect Should Know

Frank Buschmann

Siemens AG, Corporate Technology

INTERNATIONAL SOFTWARE DEVELOPMENT CONFERENCE

gotocon.com



A challenging question!

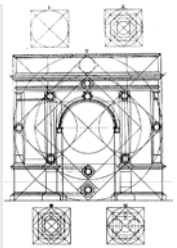
Observation

- Many architects have a sound knowledge in software engineering methods and concrete design and programming technologies, **both broad and deep**
- Yet time and again architectures and their realizations suffer from insufficient quality, regarding modularization, interactions, or non-functional quality

… even if scope and requirements are sufficiently known!

What makes architects a master of their craft?

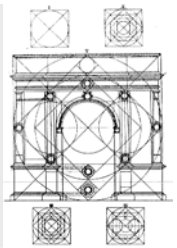
How do master architects design?



The Seven Secrets

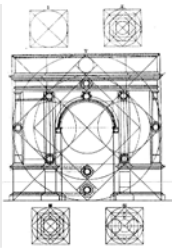
Agenda

- **Drive Design Through User Tasks**
- **Be Minimalist**
- **Ensure Visibility of Domain Concepts**
- **Use Uncertainty as a Driver**
- **Design Between Things**
- **Pay Attention To Implicit Assumptions**
- **Eat Your Own Dog Food**



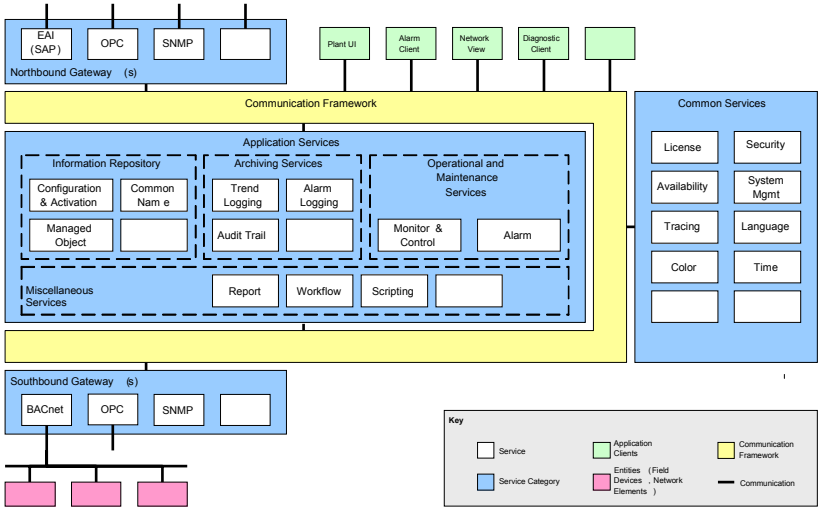
Secret One

Drive Design Through User Tasks



Unusable architectures are useless

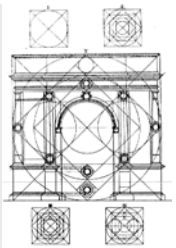
An architecture is more than a set of interacting parts, technical infrastructure, and smart design concepts!



The two key questions are:

- How well does it support **operators** in their daily work?
  - Those who live with the system?
- How well does it support **developers** in its use, realization, test, maintenance and evolution?
  - Those who live within the system?

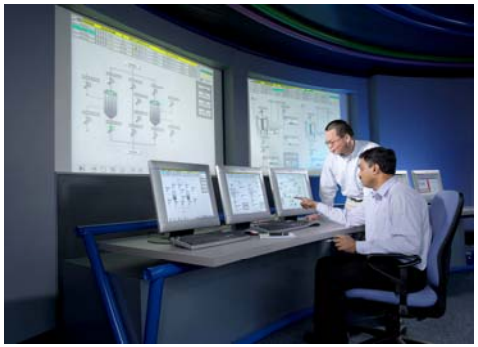
Many projects fail due to missing or balancing the two aspects!



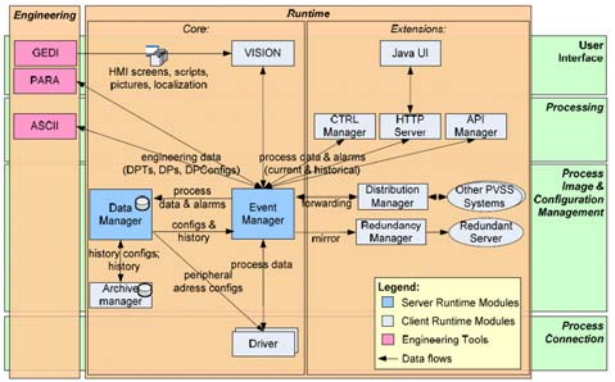
Task-oriented design

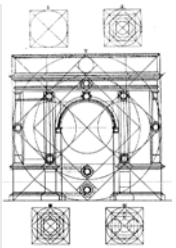
Design software systems with explicit consideration of how they will be used and how they can best support the work their users will be doing

[Larry Constantine]



- Select essential user scenarios
  - Operational scenarios: user tasks and workflows; including their quality attributes
  - Developmental scenarios: realization, adaptation, configuration, evolution, …
- Define the architecture along the selected scenarios
  - Components, workflows, interfaces, interactions, infrastructure, guiding principles
  - Focus is on the tasks, not on single functions
  - Pay attention to sensitivity and trade-off points
  - Address non-functional quality

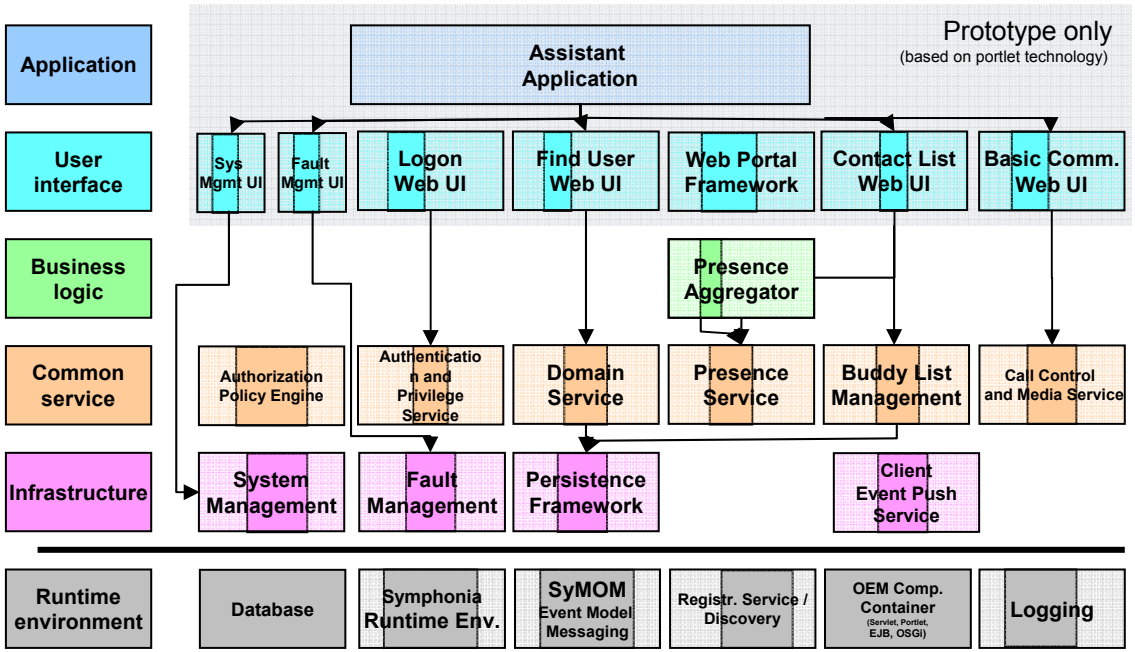




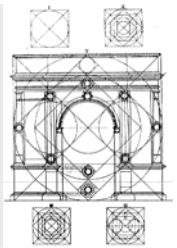
Walking skeletons

Goal of a task-driven architecture specification is to create a **walking skeleton …**

- A set of end-to-end slices through the system that correspond to architecturally significant user tasks – with users being end users and developers
- Implemented in product quality: functionality with associated quality

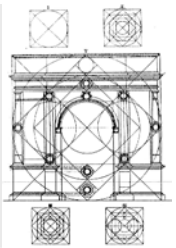


… that provides a direct feedback loop on the architecture's sustainability!



Secret Two

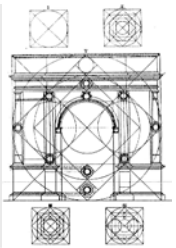
Be Minimalist



On minimalism

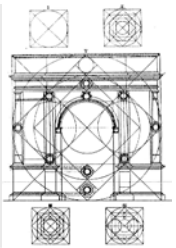
A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away

[Antoine de Saint-Exupéry]



Maximalism

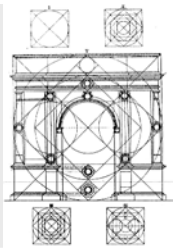
```
interface Iterator
{
    boolean set_to_first_element();
    boolean set_to_next_element();
    boolean set_to_next_nth_element(in unsigned long n) raises(…);
    boolean retrieve_element(out any element) raises(…);
    boolean retrieve_element_set_to_next(out any element, out boolean more) raises(…);
    boolean retrieve_next_n_elements
                (in unsigned long n, out AnySequence result, out boolean more) raises(…);
    boolean not_equal_retrieve_element_set_to_next(in Iterator test, out any element) raises(…);
    void remove_element() raises(…);
    boolean remove_element_set_to_next() raises(…);
    boolean remove_next_n_elements(in unsigned long n, out unsigned long actual_number) raises(…);
    boolean not_equal_remove_element_set_to_next(in Iterator test) raises(…);
    void replace_element(in any element) raises(…);
    boolean replace_element_set_to_next(in any element) raises(…);
    boolean replace_next_n_elements
                (in AnySequence elements, out unsigned long actual_number) raises(…);
    boolean not_equal_replace_element_set_to_next(in Iterator test, in any element) raises(…);
    boolean add_element_set_iterator(in any element) raises(…);
    boolean add_n_elements_set_iterator
                (in AnySequence elements, out unsigned long actual_number) raises(…);
    void invalidate();
    boolean is_valid();
    boolean is_in_between();
    boolean is_for(in Collection collector);
    boolean is_const();
    boolean is_equal(in Iterator test) raises(…);
    Iterator clone();
    void assign(in Iterator from_where) raises(…);
    void destroy();
};
```



Minimalism

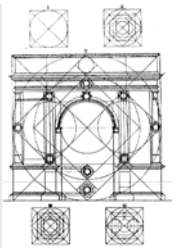
```
interface BindingIterator
{
    boolean next_one(out Binding result);
    boolean next_n(in unsigned long how_many, out BindingList result);
    void destroy();
};
```

- Clarity is often achieved by reducing clutter
  - Simpler to understand, communicate, and **test**
  - But don't encode the design or code
- Compression can come from careful abstraction
  - Compression relates to **directness** of expression
  - Abstraction concerns the **removal** of specific detail
- Abstraction is a matter of choice: the quality of abstraction relates to compression and clarity
  - Encapsulation is a vehicle for abstraction
  - What is the simplest design that possibly could work? [Ward Cunningham]



Secret Three

Ensure Visibility of Domain Concepts



Information hiding

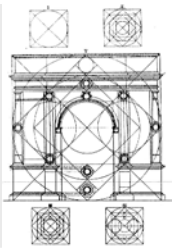
The two predominant concepts in our software

`string`

`int`

??

- What are the implied application concepts behind `string` and `int`?
- What is their intended usage contract?
- How can we ensure intention and contract are visible and enforced?



Why visibility matters – the Ariane V crash

- Velocity was represented as a 64 bit `float`
- A conversion into a 16 bit signed `integer` caused an overflow
- The current velocity of Ariane 5 was too high to be represented as a 16 bit `integer`
- Error handling was suppressed for performance reasons
- Damage 370 Mio. USD

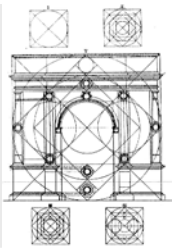
Original code fragment* (commented and reformatted)

```
-- Vertical velocity bias as measured by sensor
L_M_BV_32 :=
   TBD.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
   G_M_INFO_DERIVE(T_ALG.E_BV));

-- Check, if measured vertical velocity bias can be
-- converted to a 16 bit int. If so, then convert
if L_M_BV_32 > 32767 then
   P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
   P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
   P_M_DERIVE(T_ALG.E_BV) :=
         UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M_BV_32));
end if;

-- Horizontal velocity bias as measured by sensor
-- is converted to a 16 bit int without checking
P_M_DERIVE(T_ALG.E_BH) :=
   UC_16S_EN_16NS (TDB.T_ENTIER_16S ((1.0/C_M_LSB_BH) *
   G_M_INFO_DERIVE(T_ALG.E_BH)));
```

*Source: http://moscova.inria.fr/~levy/talks/10enslongo/enslongo.pdf

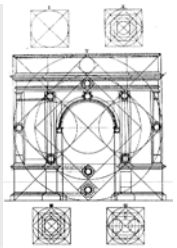


Concretion of implied concepts

- Discovery of types for values, management and control, collectives, domains, and so on
  - Implied concepts or collocated capabilities can be made more visible by recognizing these as distinct and explicit types – usage becomes type
  - Explicit types support **testability** and **design by contract**
- For example …
  - Strings for keys and codes become types in their own right, for example ISBNs, SQL statements, URLs
  - Recurring value groupings become whole objects, for example date, address, access rights

Date
Integer day, month, year
String getDate() Integer getDayInMonth() Integer getMonth() Integer getYear()

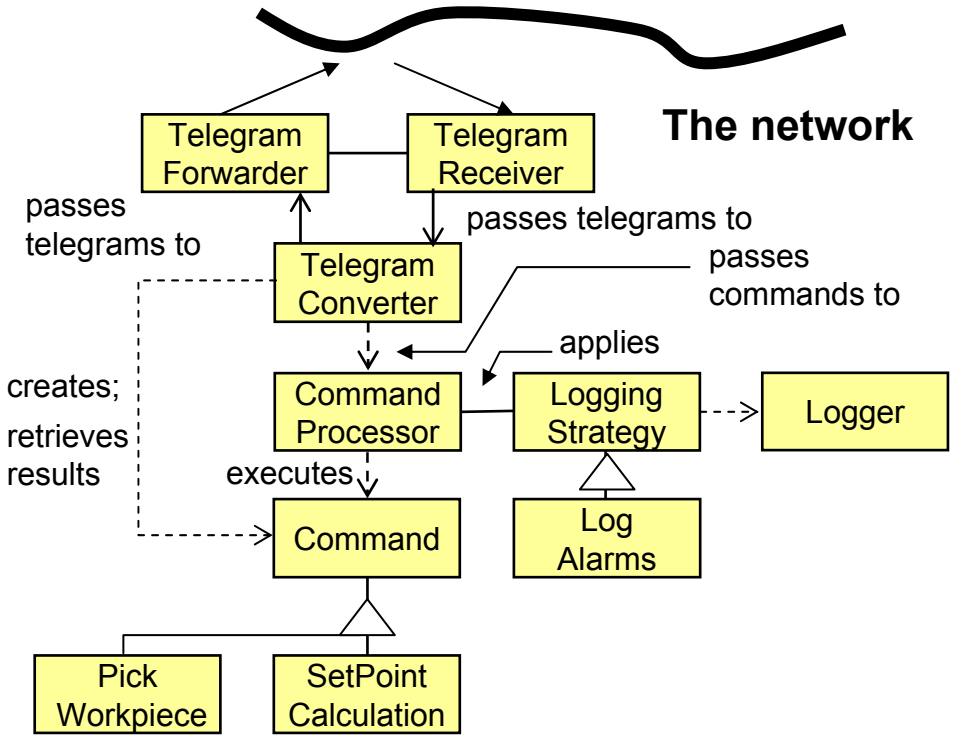
ISBN
String isbn
String asString()



Expressive architecture

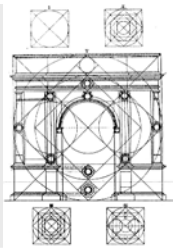
Visibility in a software architecture amounts to expressiveness

- Components and their relationships should be related by names that reflect their nature
- Components should have cohesive responsibilities, contractual interfaces and explicit relationships
- By looking through the artifacts, both the essence and detail should be apparent



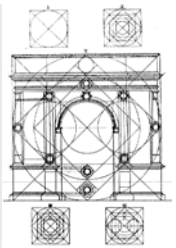
A (simplified) design for a telegram handler in a factory automation system

Expressive designs are easier to understand, communicate, realize, test, and review



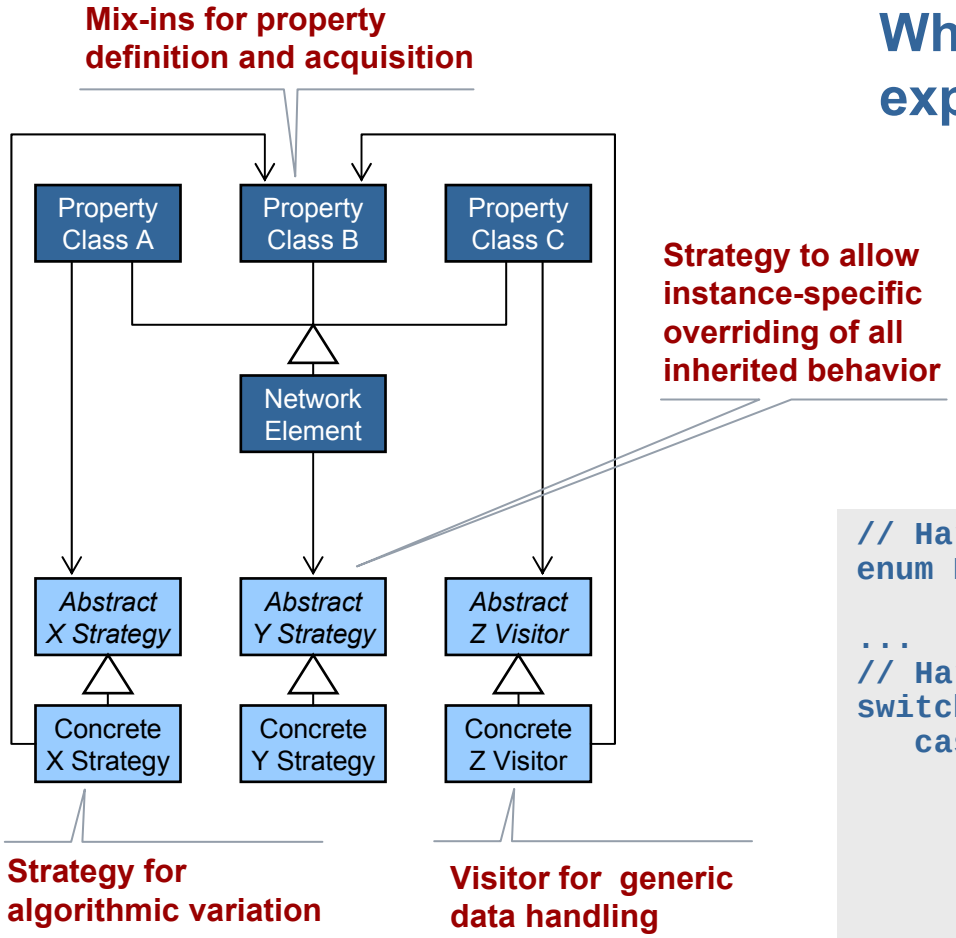
Secret Four

Use Uncertainty as a Driver



The indecisive architect

Uncertainty is no excuse for indecisiveness and escape!



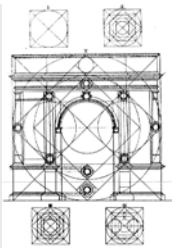
When architects cannot agree on an explicit domain model …

… and escape in genericity

Someone else will decide and make it concrete!!

```
// Hard-coded enumeration of domain types
enum Node_Type {A, B, C, D, E}; // Anonymized

...
// Hard-coded instance configuration
switch (node->type()) {
   case A:
      node->add_property
         (new Persistance(new Data_Visitor_A));
      node->add_property
         (new Network_Connectivity());
      ...
```



Use uncertainty as a driver

The most interesting thing is not actually the choice between A and B, but the fact that there is a choice between A and B [Kevlin Henney]

Limit uncertainty scope

Localize, isolate, and encapsulate the fact there is a choice

- To avoid rippling effects on other system parts
- Works well for structural, algorithmic, implementation, and many technology choices; difficult for cross-cutting concerns

Make uncertainty explicit

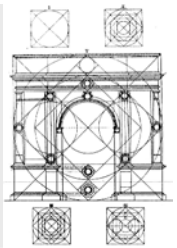
Decide explicitly (!) to not decide now (!!)

Drive uncertainty resolution

Take concrete action to (iteratively) drive decision

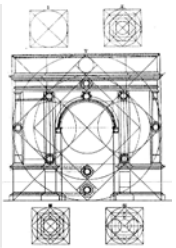
- Spikes to sharpen requirements and technology understanding
- Spikes to get feedback on each option's sustainability
- Scenarios in a walking skeleton to stress operational consequences of the most promising option
- Active Design Reviews to explore the developmental consequences of the most promising option

Decide



Secret Five

Design Between Things



An all to common interface tale

The interface signature when released

```
/// Manage business object metadata (tags) in a key value map
class TagManager {
 ...
 /// Assign a string "value" to the identifier "tagName"
 public void set(string tagName, string value);
 ...
}
```

Tag management did not include tag removal

Developers discovered need for tag removal …

… but changing a released interface is difficult

The solution

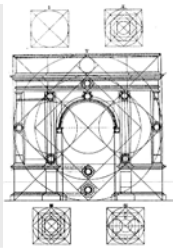
1. Change `set()` implementation to handle both `set()` and `remove()`
2. Convention to use a prefix `"r:"` in `tagName` to indicate removal

```
/// Call to set a tag
tag.set(tagName, value);

/// Call to remove a tag
tag.set("r:" + tagName, new string());
```

Tag removal added without interface modification …

… but at the cost of contract violation

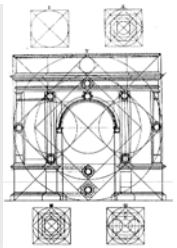


Be where things meet

The architect's main territory is between things, where they meet and hurt: Interfaces, Interactions, Integration.

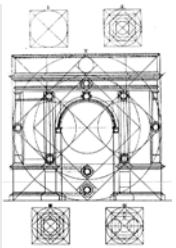
Interfaces	▪ Complete, meaningful ▪ Role-specific, expressive, usable ▪ Defined contract, stability
Interaction	▪ Simple, meaningful, direct, efficient ▪ Quality of Service (reliable, fast, scalable, secure, configurable, …) ▪ Task-oriented, end-to-end quality
Integration	▪ Systems integration, plant integration, HW / SW integration ▪ Application integration, service integration, process-level integration, data integration, UI integration, device integration

Deficiencies in interfaces, interactions, and integration tend to show up later in the SW lifecycle than modularization and implementation issues: during system integration, system test, roll out, operation – thus their resolution is costly!



Secret Six

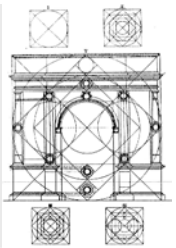
Pay Attention To Implicit Assumptions



The cost of the unspecified (1)

Imagine you are asked to maintain this code …

- Where do you likely feel comfortable?
- Where is the code smelling badly?
- Where do you think challenges occur and maintenance is expensive?



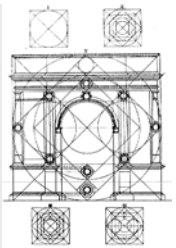
The cost of the unspecified (2)

The business success or acceptance of software often depends on qualities that are rarely explicitly stated

85% of SW Lifecycle costs are devoted to maintenance and evolution

Developers spend **50% of their time** in the process of understanding code

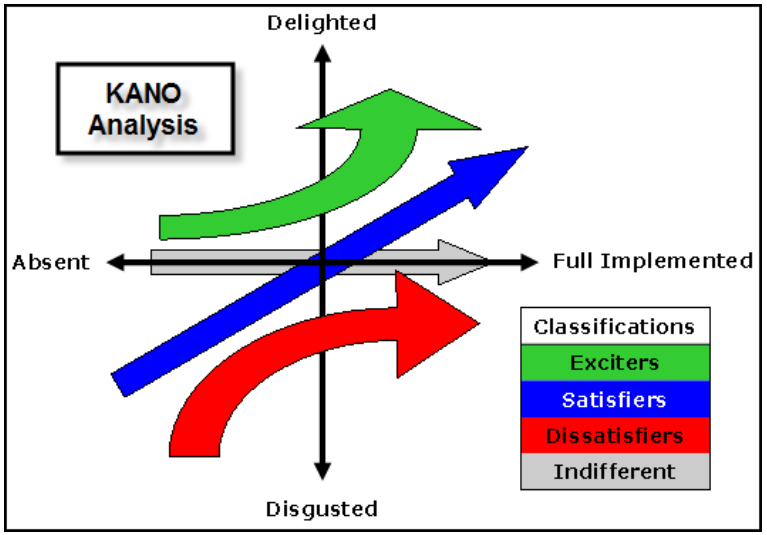
Koskinen, University of Jyväskylä, Finland
http://users.jyu.fi/~koskinen/smcosts.htm



Watch out for dissatisfiers

Pay attention to implicit assumptions: features or qualities that are expected and create trouble if not present, but rarely expressed explicitly!

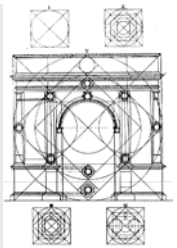
- Functionality of competitor or previous generation systems
- Performance
- Robustness
- Maintainability
- … [check your spec]



Address expected features prominently, constantly, and from the very beginning

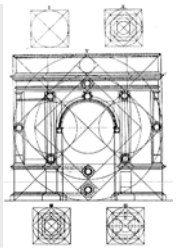
- To make them visible and explicit in your architecture
- To avoid costly late changes and non-conformance costs
- To avoid customer dissatisfaction

A KANO helps identifying expected requirements – which tend to correspond to dissatisfiers



Secret Seven

Eat Your Own Dog Food



Theory and practice can differ!

Even the smartest design concepts can create trouble!

- Introduced with care and intention; well understood by architect
- Misunderstood by all others; or hard to implement

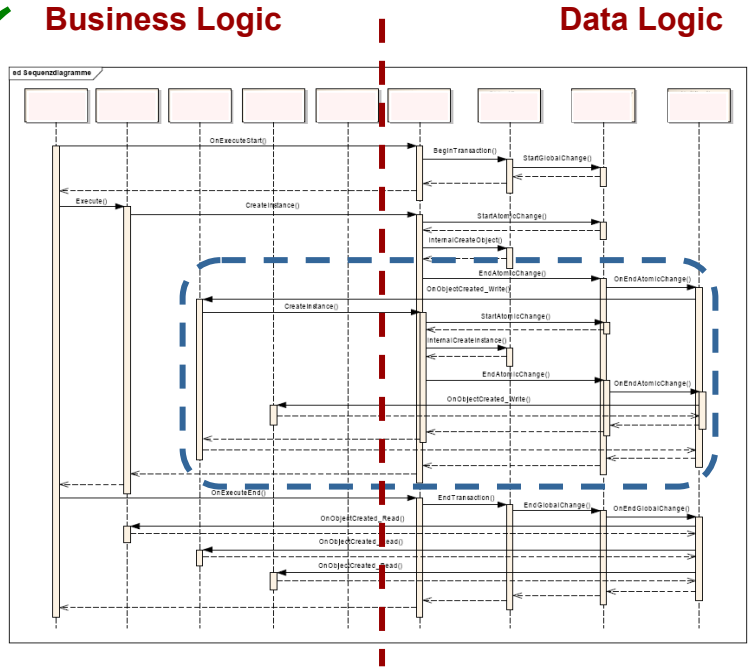
Architect intention

Problem

General data handling logic, e.g., copy, move, needs application-specific extensions

Solution

- Interceptor framework in data handling logic;
- Self-contained interceptors provide local extension logic



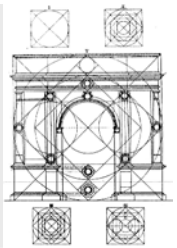
Actual use

Interceptors

- Were not closed but called back “normal” application logic
- Application logic issued nested calls to data logic

Effect

- Oscillating control flow between business logic and data logic
- Circular, uncontrollable, unstable call chains



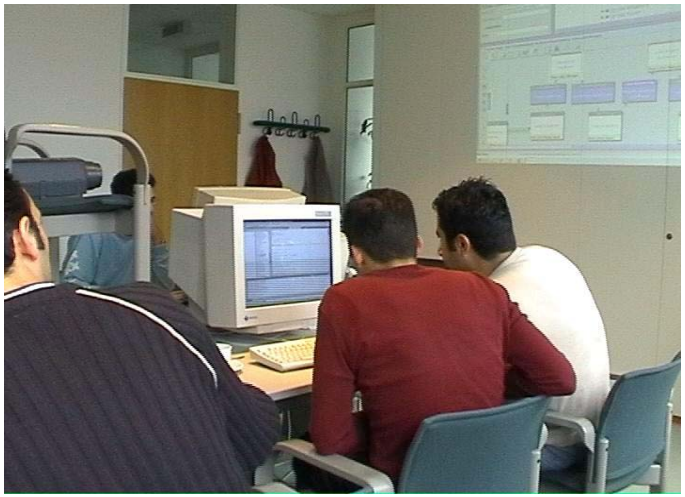
Architect also implements!

Eat your own dog food! Actively participate in implementation!

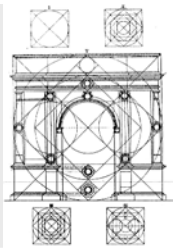
- To experience the consequences of your own designs!
- To communicate your intention, and minimize misunderstanding
- To discover the devil in the detail

But, don't loose yourself in code!

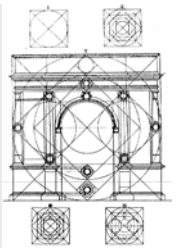
- Prefer pair programming over lone coding
  - To cover all system parts
  - To reach all developers
  - To avoid being overloaded or on a critical path
- Focus on essential aspects or scenarios
  - To ensure developer habitability
  - To ensure concept sustainability
- Write tests and conduct active design reviews
  - To guide development
  - To test interfaces, interaction, integration



Seven Secrets Every Architect Should Know



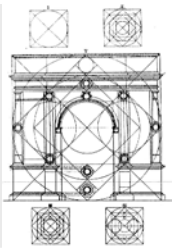
In Retrospect



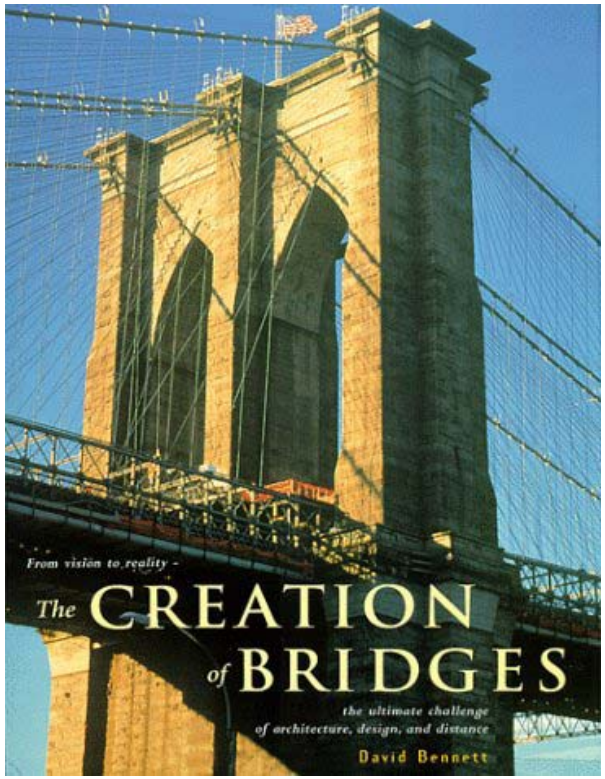
In retrospect

The seven practices complement an architect's knowledge, technical experience, and design skills regarding

- **Communication to stakeholders**
  - Key measures: explicit attention to their interests, making interests visible and tangible, e.g., through walking skeletons or attention to uncertainty
- **Economic architecture**
  - Key measure: strict adherence to KISS principle
- **Understanding that code matters**
  - Key measure: explicit consideration of developer needs, early coding, presence and participation in coding
- **Seek for feedback**
  - Key measure: visibility and explicit attention to challenges, both known and unknown



A departing thought



Structural engineering is the science and art of designing and making, with economy and elegance, buildings, bridges, frameworks, and other similar structures so that they can safely resist the forces to which they may be subjected.

[The Institution of Structural Engineers]