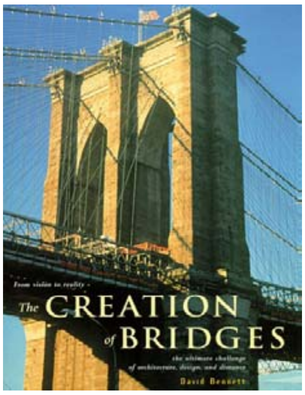
Five Considerations for Software Developers

Frank Buschmann
Siemens AG, Corporate Technology
`Frank.Buschmann@siemens.com`

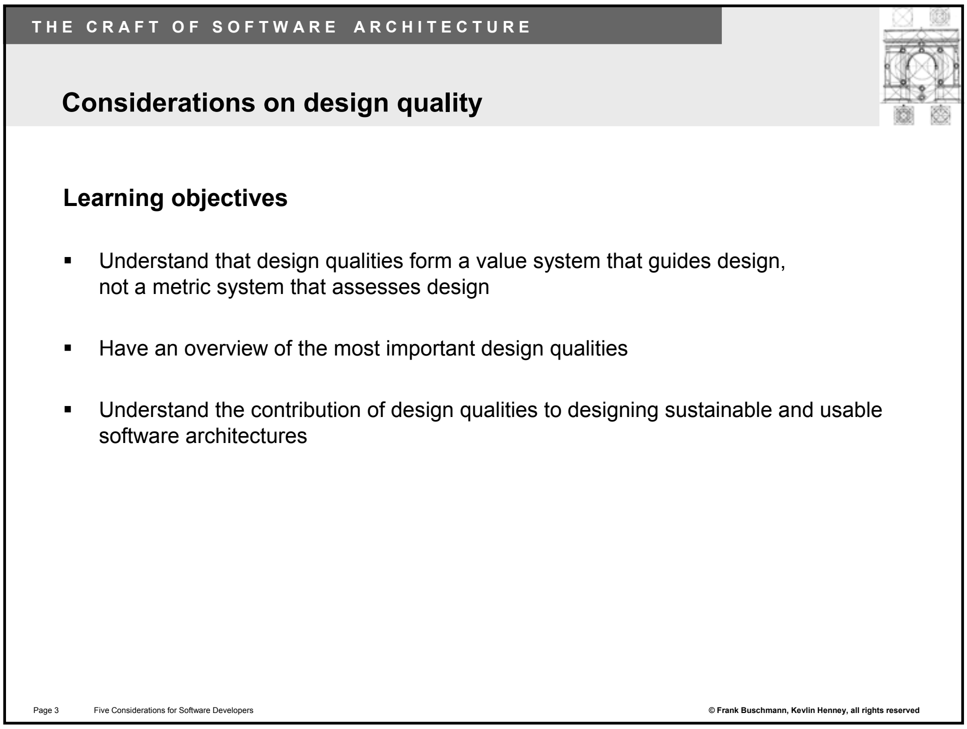
Kevlin Henney
Curbralan Ltd.
`Kevlin@curbralan.com`

Designing with economy and elegance


From vision to reality -
The CREATION of BRIDGES
the ultimate challenge of architecture, design, and distance
David Bennett

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.

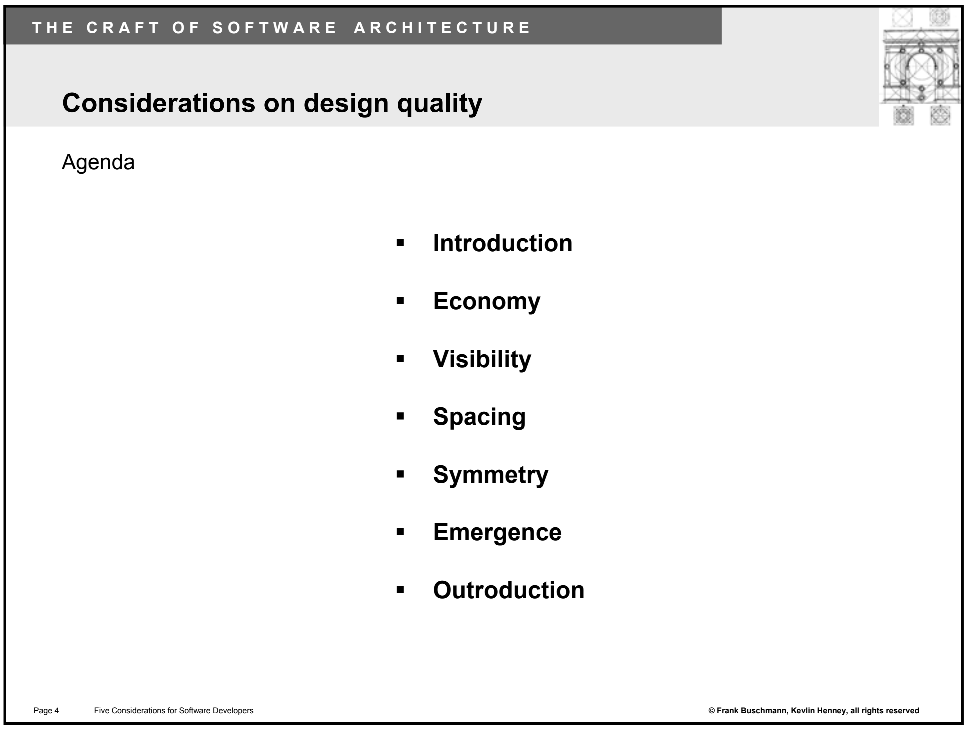
[The Institution of Structural Engineers]



Considerations on design quality

Learning objectives

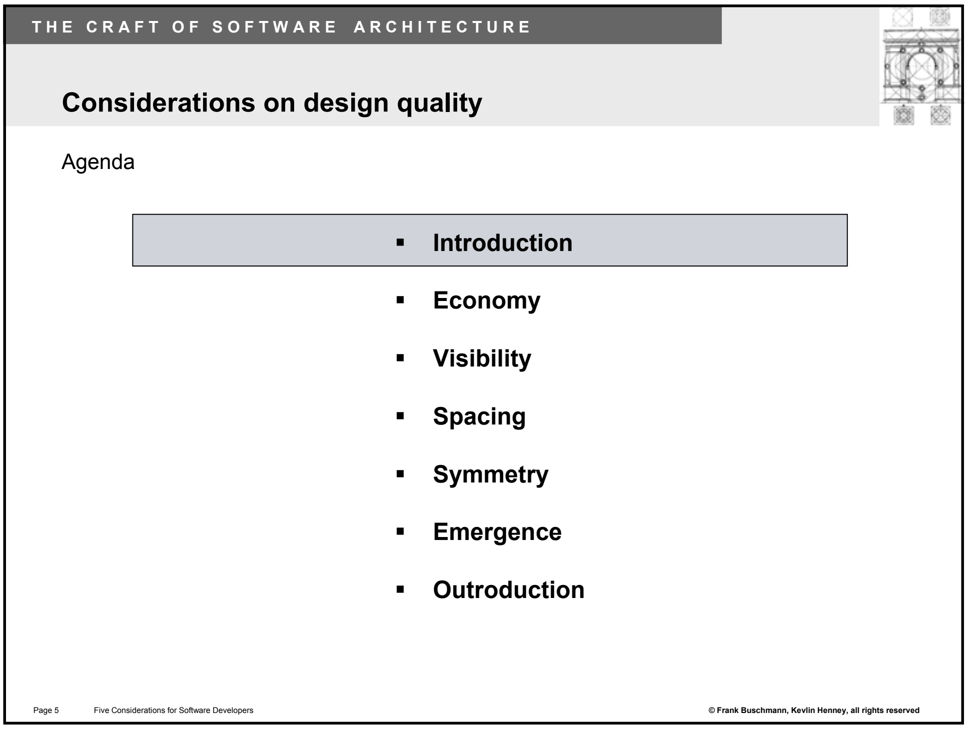
- Understand that design qualities form a value system that guides design, not a metric system that assesses design
- Have an overview of the most important design qualities
- Understand the contribution of design qualities to designing sustainable and usable software architectures



Considerations on design quality

Agenda

- **Introduction**
- **Economy**
- **Visibility**
- **Spacing**
- **Symmetry**
- **Emergence**
- **Outroduction**

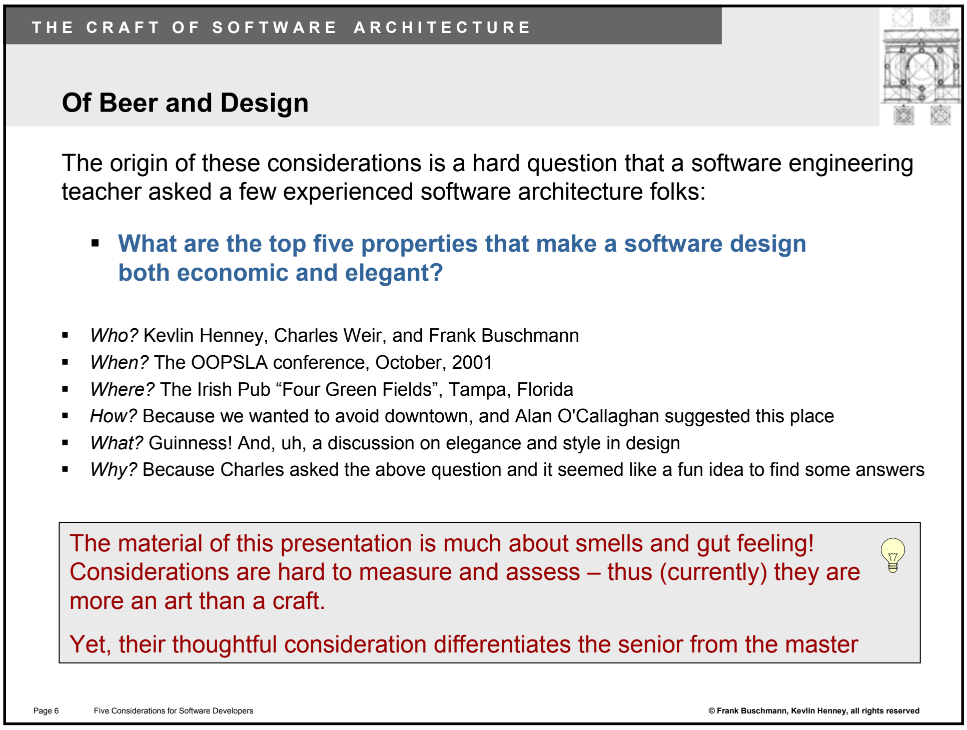


THE CRAFT OF SOFTWARE ARCHITECTURE

Considerations on design quality

Agenda

- **Introduction**
- **Economy**
- **Visibility**
- **Spacing**
- **Symmetry**
- **Emergence**
- **Outroduction**



THE CRAFT OF SOFTWARE ARCHITECTURE

Of Beer and Design

The origin of these considerations is a hard question that a software engineering teacher asked a few experienced software architecture folks:

- **What are the top five properties that make a software design both economic and elegant?**

- *Who?* Kevlin Henney, Charles Weir, and Frank Buschmann
- *When?* The OOPSLA conference, October, 2001
- *Where?* The Irish Pub “Four Green Fields”, Tampa, Florida
- *How?* Because we wanted to avoid downtown, and Alan O'Callaghan suggested this place
- *What?* Guinness! And, uh, a discussion on elegance and style in design
- *Why?* Because Charles asked the above question and it seemed like a fun idea to find some answers

The material of this presentation is much about smells and gut feeling! Considerations are hard to measure and assess – thus (currently) they are more an art than a craft.

Yet, their thoughtful consideration differentiates the senior from the master

Considering considerations

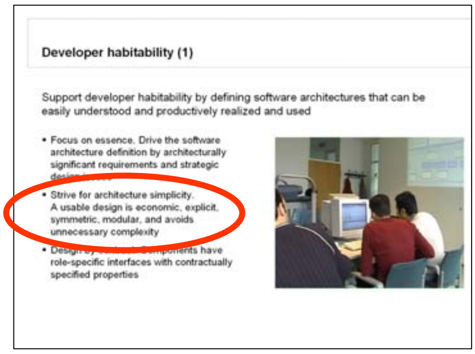
- A consideration is not a rule
  - And it is also weaker than the conventional notion of a recommendation
  - It is … a consideration
- A consideration takes a point of view
  - It may be general, it may be specific
- A system of considerations can offer a coherent and unified set of views
  - Together they can guide recommendations

Caveat: A quality architecture exhibits most, if not all, of the design qualities we present here, but the counter conclusion is plain wrong: a software architecture that exposes the qualities is not necessarily good!

Design quality considerations form a value system

Considerations on design quality form a **value system** that helps **guiding** the definition of software architectures

- The considerations on design quality are well integrated with the framework for architecture-centric software engineering of platforms and product-lines:
  - Foundation in requirements, and thus the business case
  - Strong focus on architecture usability, in particular developer habitability
  - Guide quality assurance and testing activities, in particular architecture reviews and design rule checking
  - Support communication of an architecture to stakeholders

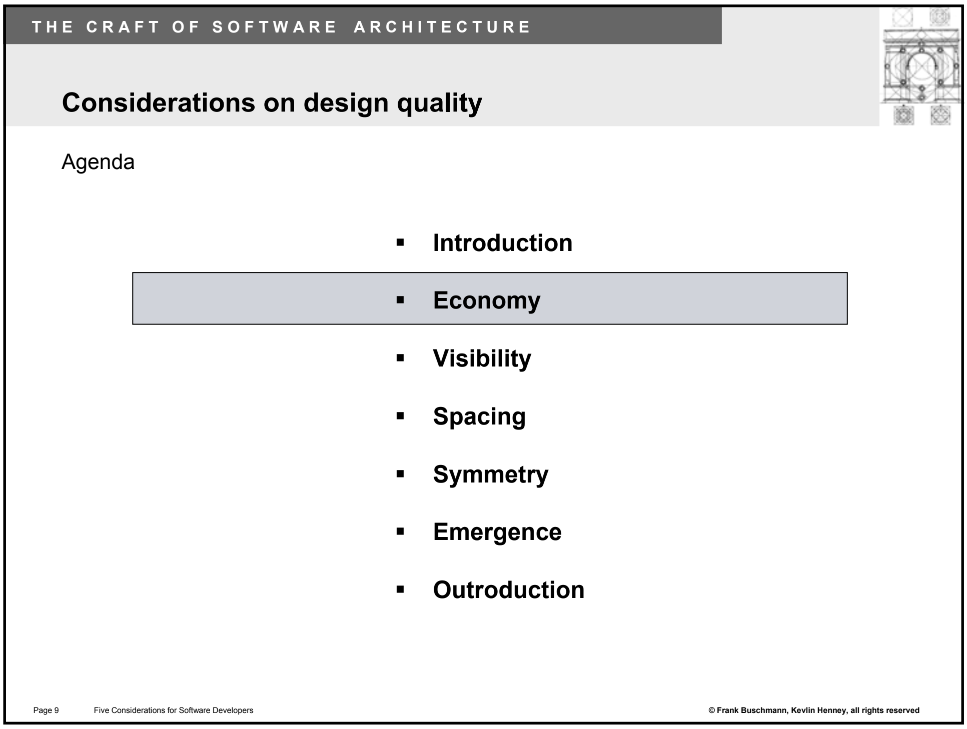


Developer habitability (1)

Support developer habitability by defining software architectures that can be easily understood and productively realized and used

- Focus on essence. Drive the software architecture definition by architecturally significant requirements and strategic design issues
- Strive for architecture simplicity. A usable design is economic, explicit, symmetric, modular, and avoids unnecessary complexity
- Design by contract. Components have role-specific interfaces with contractually specified properties

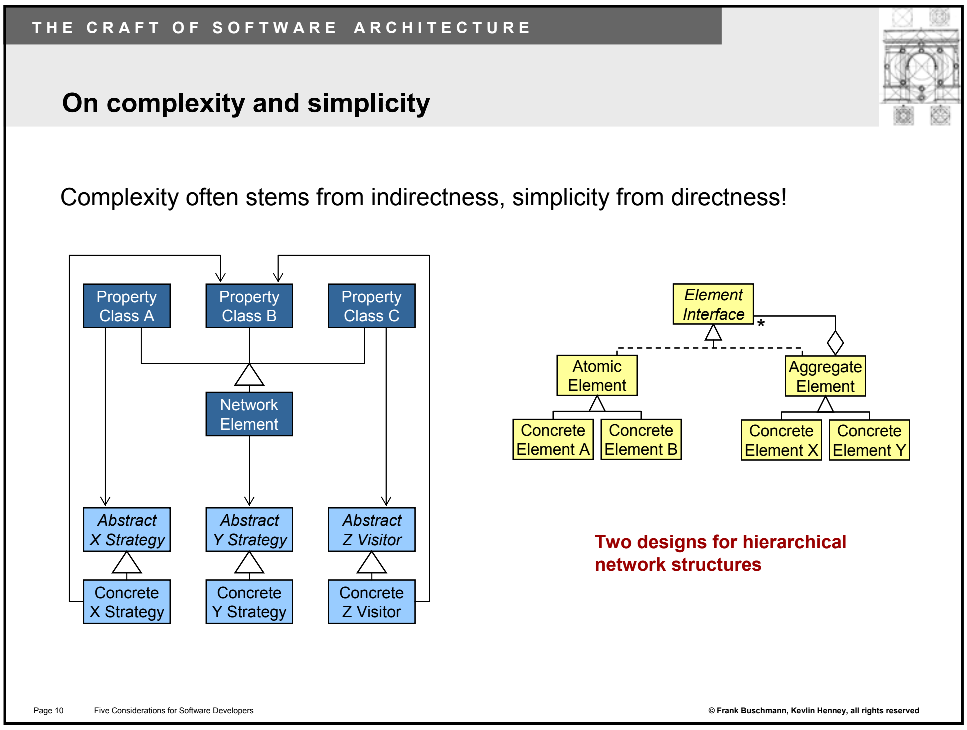
From Frank's Notes on Software Architecture Tutorial (Friday, Oct. 3)



Considerations on design quality

Agenda

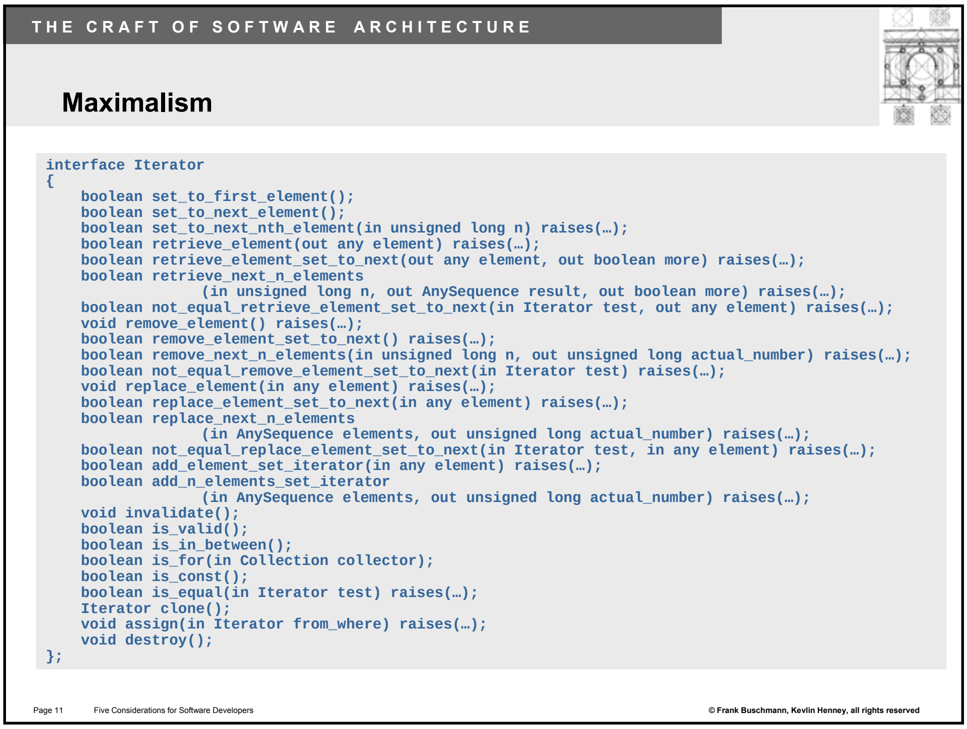
- Introduction
- **Economy**
- Visibility
- Spacing
- Symmetry
- Emergence
- Outroduction



On complexity and simplicity

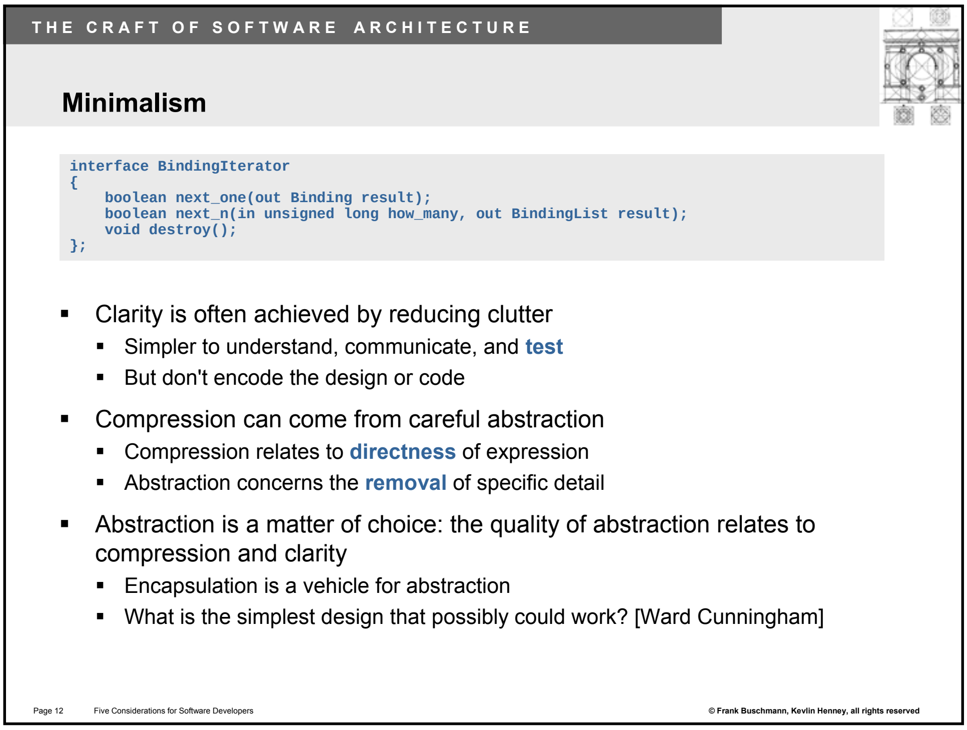
Complexity often stems from indirectness, simplicity from directness!

Two designs for hierarchical network structures



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

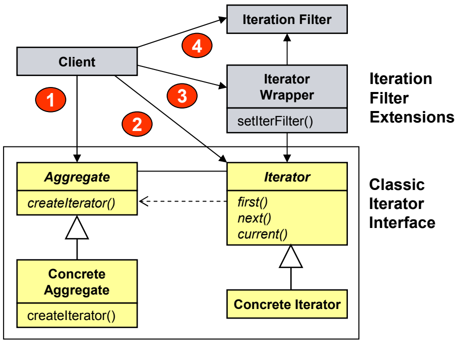
```
interface BindingIterator
{
    boolean next_one(out Binding result);
    boolean next_n(in unsigned long how_many, out BindingList result);
    void destroy();
};
```

- 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]

Designing towards requirements

A key to economy: design towards **requirements** and responsibilities, not implementations!

- Accidental complexity – the counter measure to simplicity – is often a result of programming towards, or in terms of, implementations rather than requirements
- Each design decision, therefore, should have a clear purpose that is founded in the requirements for the system, but not in the technologies used or in existing implementations



Test-first development supports designing towards requirements

Adding filtered iteration on top of the original Iterator implementation adds accidental complexity.

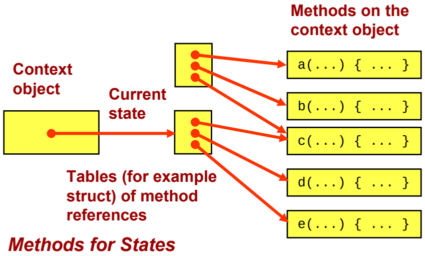
Adequate design solutions

Another key to economy: avoid the hammer-nail syndrome!

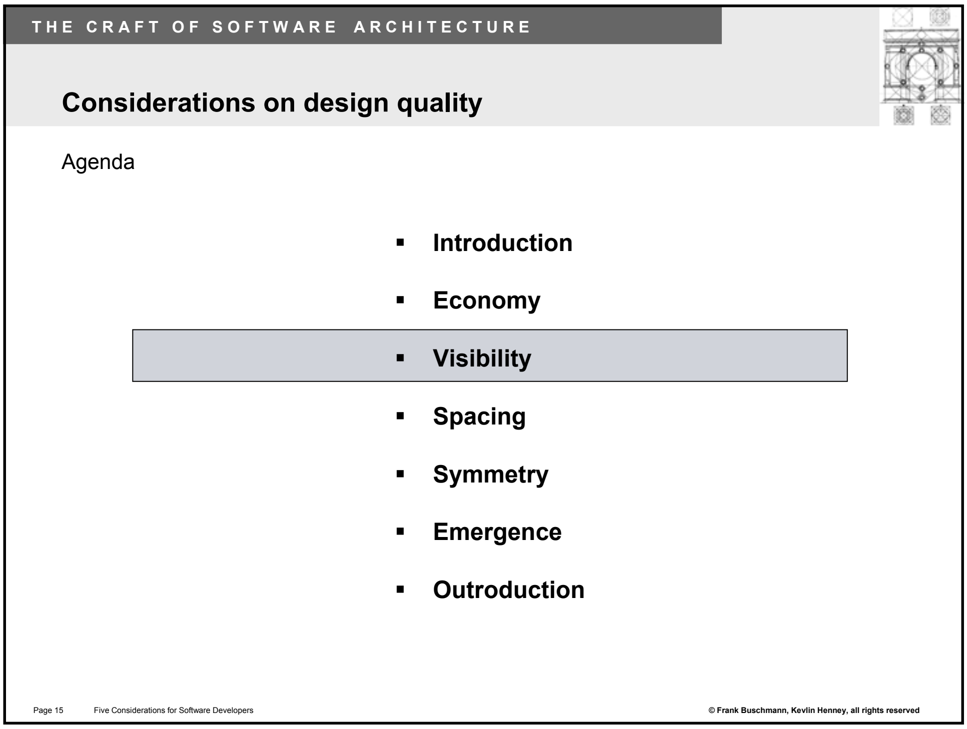
- In real-world practice, there is often more than one way to resolve a given problem
- Even if a particular way is good by itself, it still may not be appropriate for the problem under consideration
- Selecting and implementing a solution that is “just good enough” to resolve the problem or requirements at hand is fundamental for simple designs
- Patterns and other design tactics offer “catalogs of choices” for addressing recurring design problems

Many designers use the (Objects) for State(s) pattern from the Gang-of-Four to realize modal behavior. For most such situations this solution is overly complex, however, since state is encapsulated into objects.

Methods for States is often a less complex and resource-saving alternative



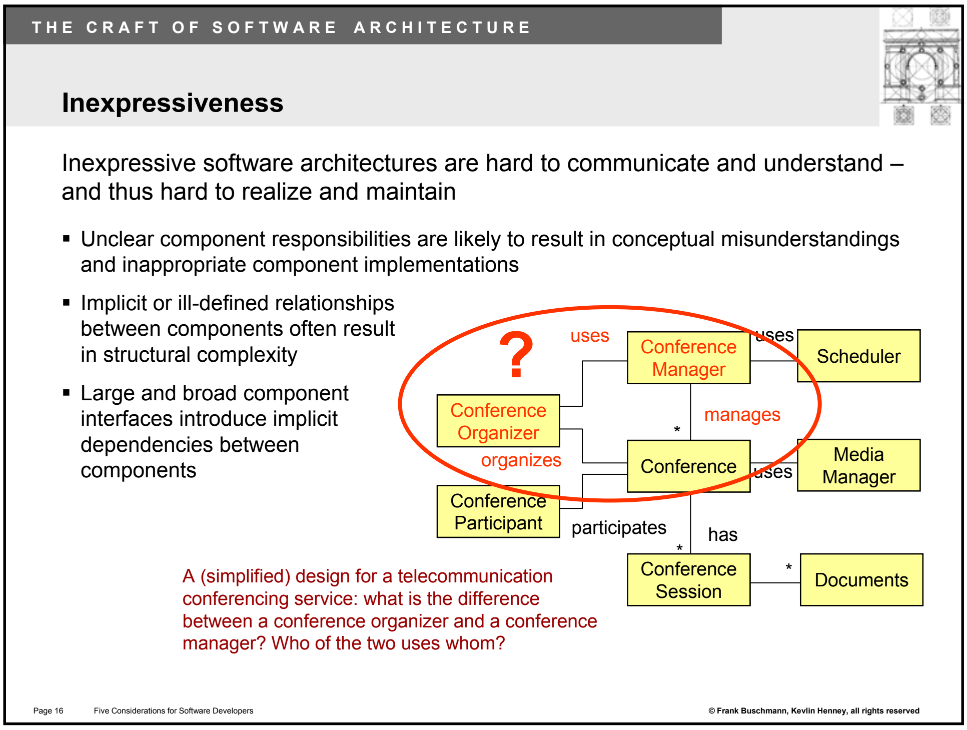
Methods for States



Considerations on design quality

Agenda

- **Introduction**
- **Economy**
- **Visibility**
- **Spacing**
- **Symmetry**
- **Emergence**
- **Outroduction**



Inexpressiveness

Inexpressive software architectures are hard to communicate and understand – and thus hard to realize and maintain

- Unclear component responsibilities are likely to result in conceptual misunderstandings and inappropriate component implementations
- Implicit or ill-defined relationships between components often result in structural complexity
- Large and broad component interfaces introduce implicit dependencies between components

?

uses

Conference Manager

uses

Scheduler

Conference Organizer

manages

*

organizes

Conference

uses

Media Manager

Conference Participant

participates

has

*

Conference Session

*

Documents

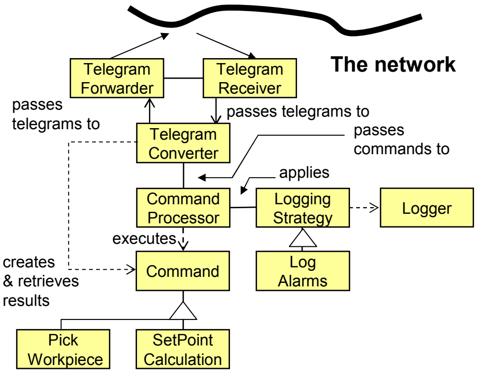
A (simplified) design for a telecommunication conferencing service: what is the difference between a conference organizer and a conference manager? Who of the two uses whom?

Expressive architecture

Visibility in a software architecture amounts to expressiveness

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

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



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

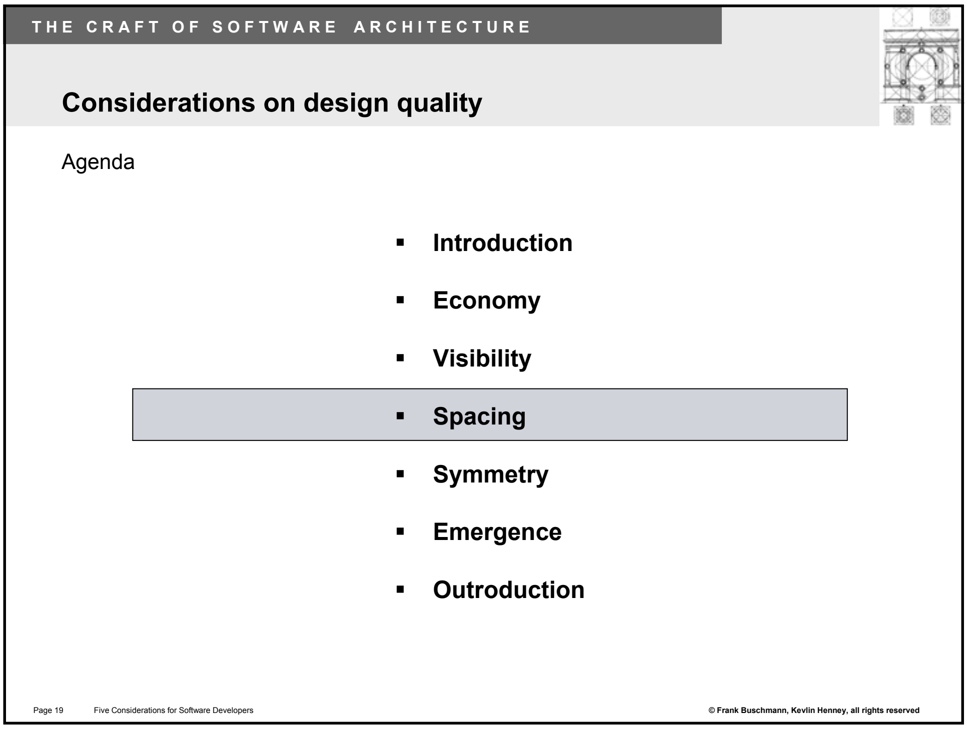
Concretion of implied concepts

- 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()

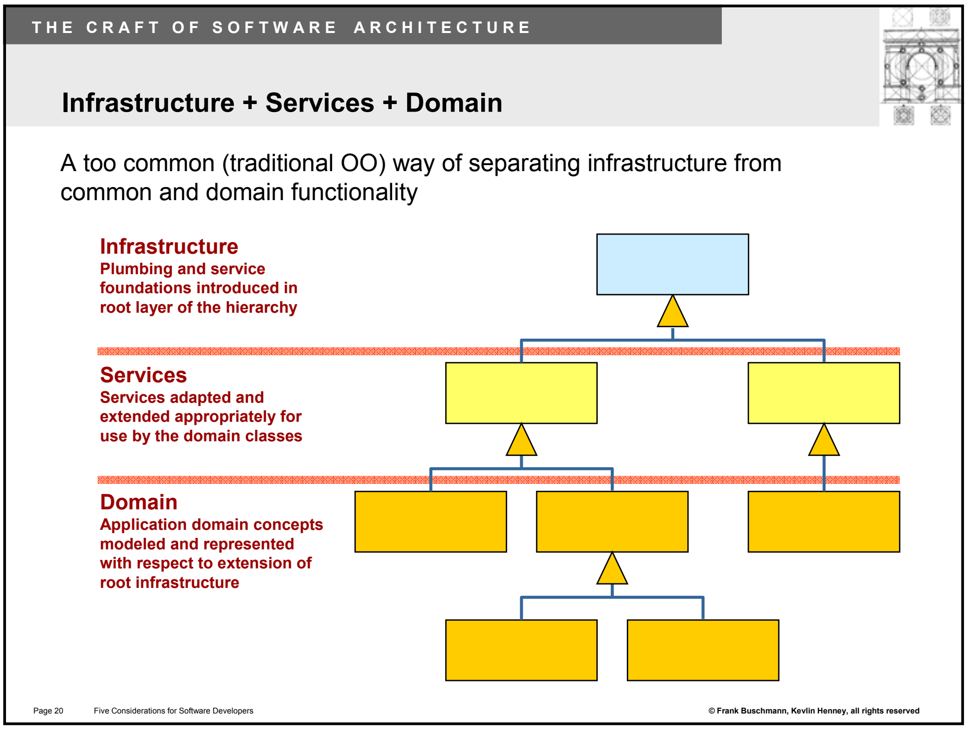
Examples of types representing real-world concepts



Considerations on design quality

Agenda

- Introduction
- Economy
- Visibility
- **Spacing**
- Symmetry
- Emergence
- Outroduction



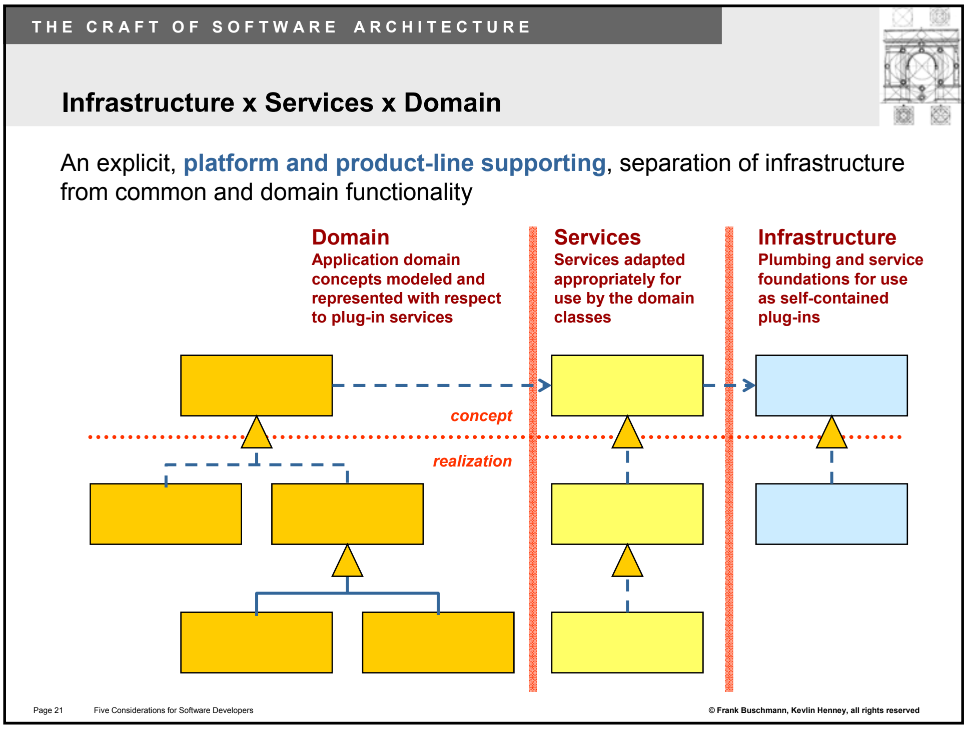
Infrastructure + Services + Domain

A too common (traditional OO) way of separating infrastructure from common and domain functionality

Infrastructure
Plumbing and service foundations introduced in root layer of the hierarchy

Services
Services adapted and extended appropriately for use by the domain classes

Domain
Application domain concepts modeled and represented with respect to extension of root infrastructure



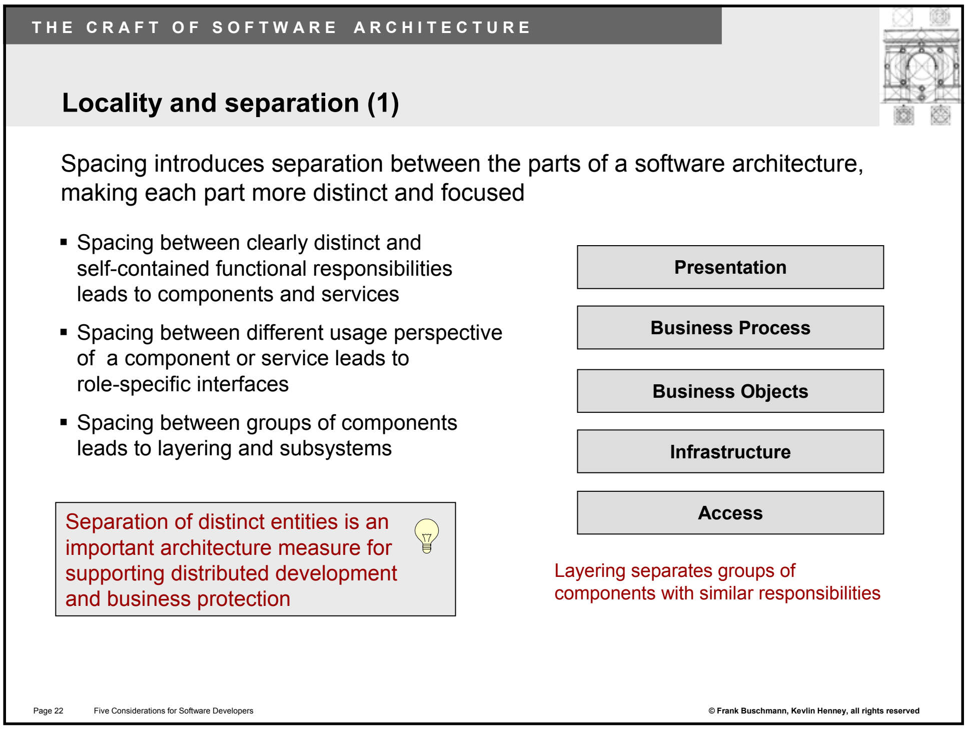
Infrastructure x Services x Domain

An explicit, **platform and product-line supporting**, separation of infrastructure from common and domain functionality

Domain
Application domain concepts modeled and represented with respect to plug-in services

Services
Services adapted appropriately for use by the domain classes

Infrastructure
Plumbing and service foundations for use as self-contained plug-ins



Locality and separation (1)

Spacing introduces separation between the parts of a software architecture, making each part more distinct and focused

- Spacing between clearly distinct and self-contained functional responsibilities leads to components and services
- Spacing between different usage perspective of a component or service leads to role-specific interfaces
- Spacing between groups of components leads to layering and subsystems

Separation of distinct entities is an important architecture measure for supporting distributed development and business protection

Presentation

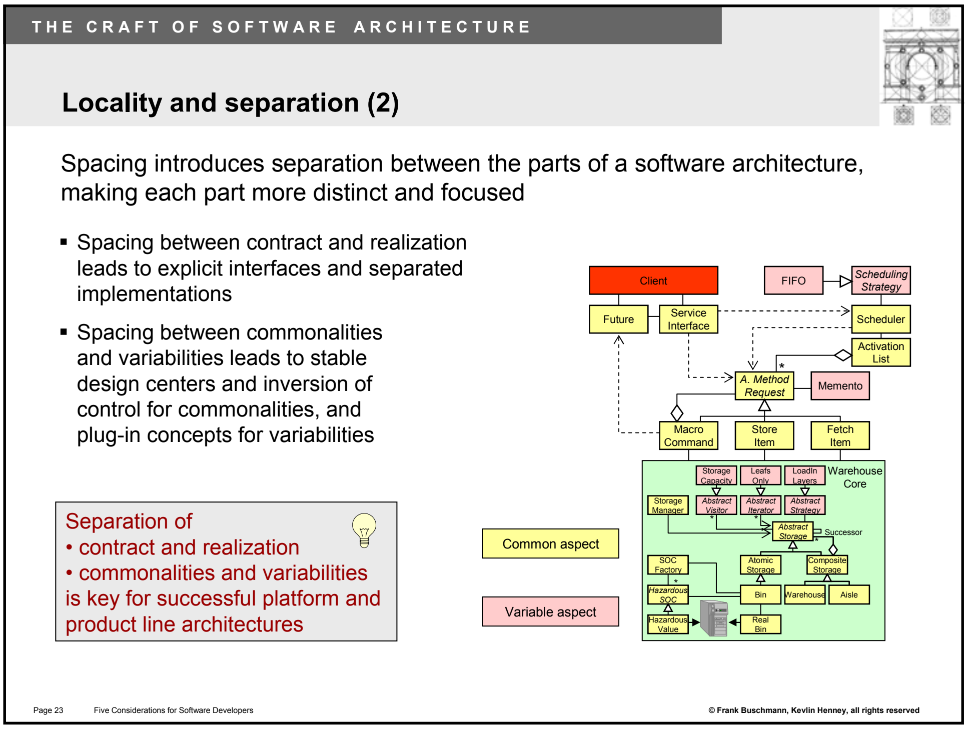
Business Process

Business Objects

Infrastructure

Access

Layering separates groups of components with similar responsibilities

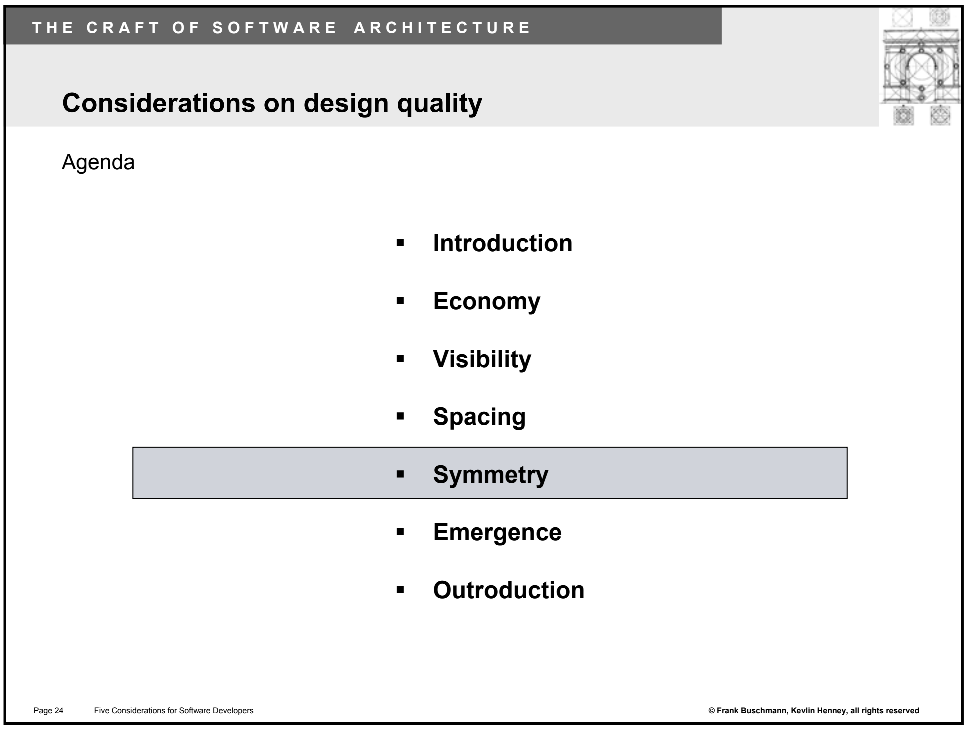


Locality and separation (2)

Spacing introduces separation between the parts of a software architecture, making each part more distinct and focused

- Spacing between contract and realization leads to explicit interfaces and separated implementations
- Spacing between commonalities and variabilities leads to stable design centers and inversion of control for commonalities, and plug-in concepts for variabilities

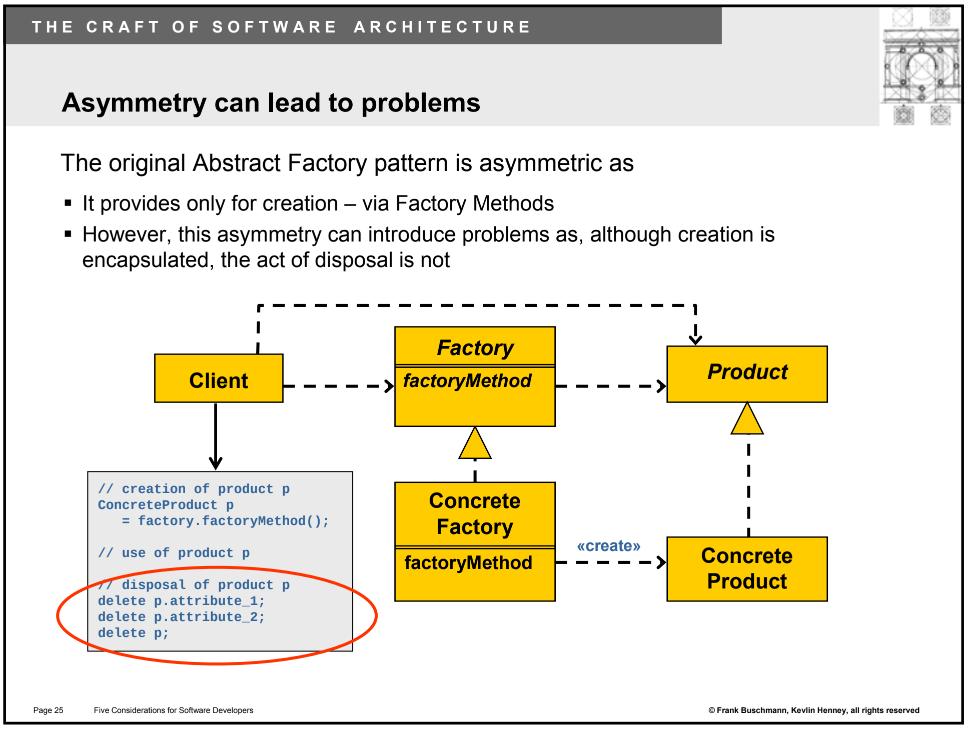
Separation of
• contract and realization
• commonalities and variabilities
is key for successful platform and product line architectures



Considerations on design quality

Agenda

- **Introduction**
- **Economy**
- **Visibility**
- **Spacing**
- **Symmetry**
- **Emergence**
- **Outroduction**



Asymmetry can lead to problems

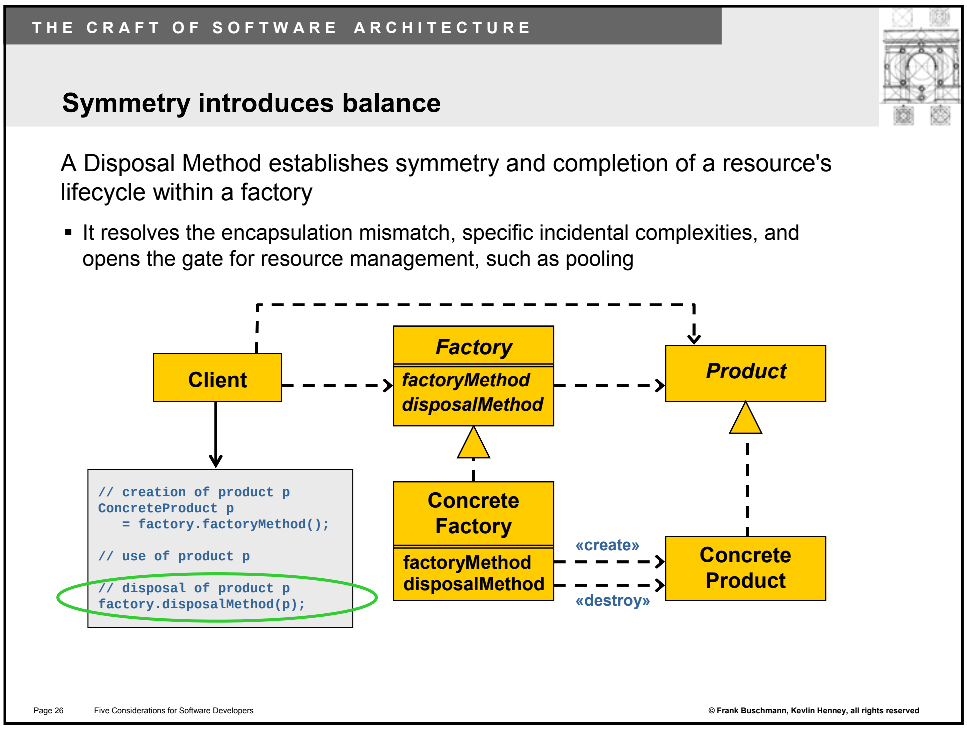
The original Abstract Factory pattern is asymmetric as

- It provides only for creation – via Factory Methods
- However, this asymmetry can introduce problems as, although creation is encapsulated, the act of disposal is not

```
// creation of product p
ConcreteProduct p
   = factory.factoryMethod();

// use of product p

// disposal of product p
delete p.attribute_1;
delete p.attribute_2;
delete p;
```



Symmetry introduces balance

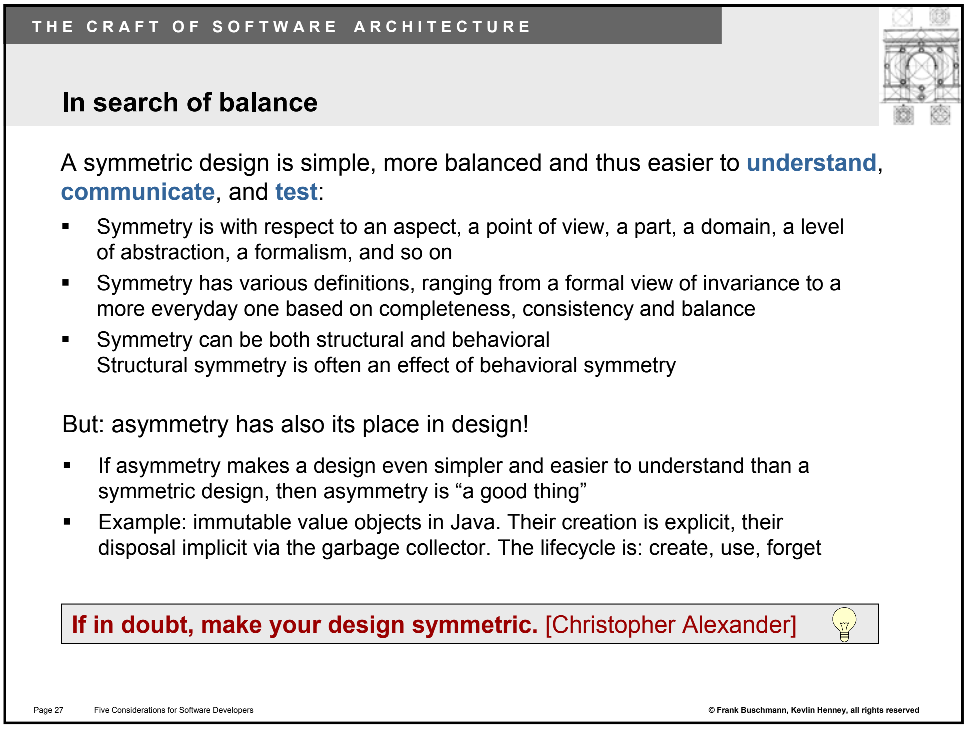
A Disposal Method establishes symmetry and completion of a resource's lifecycle within a factory

- It resolves the encapsulation mismatch, specific incidental complexities, and opens the gate for resource management, such as pooling

```
// creation of product p
ConcreteProduct p
   = factory.factoryMethod();

// use of product p

// disposal of product p
factory.disposalMethod(p);
```



In search of balance

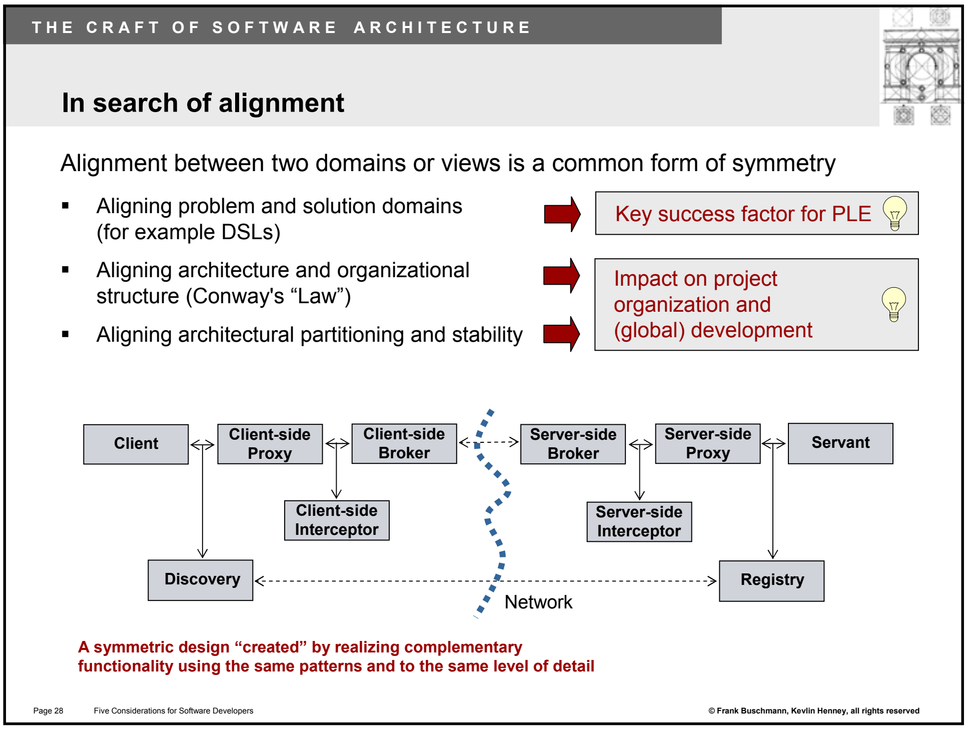
A symmetric design is simple, more balanced and thus easier to **understand**, **communicate**, and **test**:

- Symmetry is with respect to an aspect, a point of view, a part, a domain, a level of abstraction, a formalism, and so on
- Symmetry has various definitions, ranging from a formal view of invariance to a more everyday one based on completeness, consistency and balance
- Symmetry can be both structural and behavioral
  Structural symmetry is often an effect of behavioral symmetry

But: asymmetry has also its place in design!

- If asymmetry makes a design even simpler and easier to understand than a symmetric design, then asymmetry is “a good thing”
- Example: immutable value objects in Java. Their creation is explicit, their disposal implicit via the garbage collector. The lifecycle is: create, use, forget

If in doubt, make your design symmetric. [Christopher Alexander]



In search of alignment

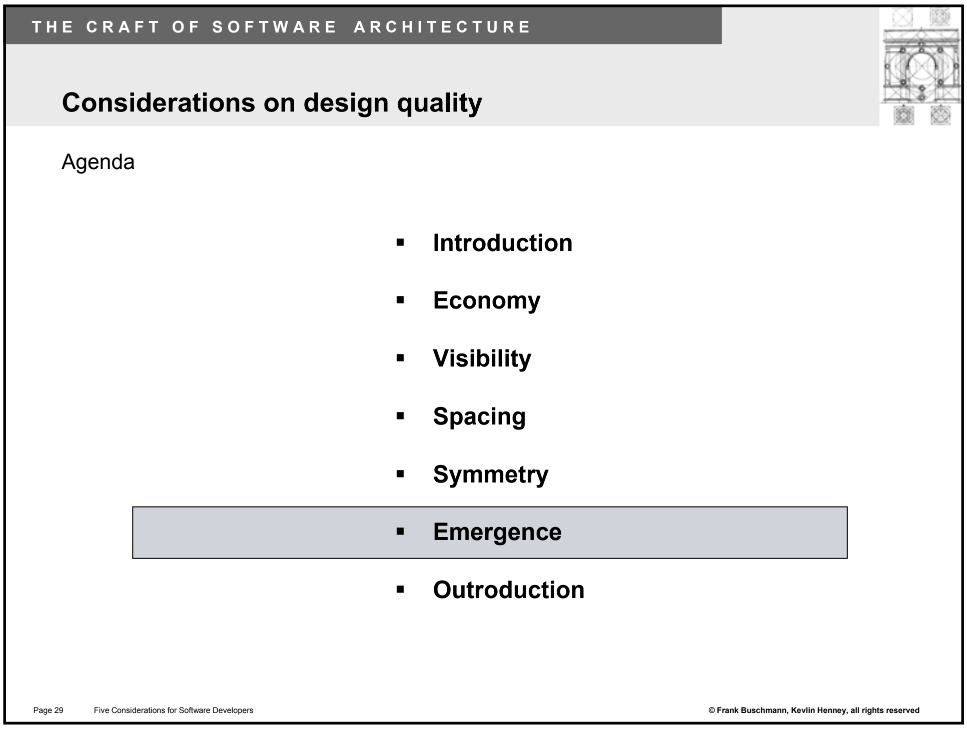
Alignment between two domains or views is a common form of symmetry

- Aligning problem and solution domains (for example DSLs)
- Aligning architecture and organizational structure (Conway's “Law”)
- Aligning architectural partitioning and stability

Key success factor for PLE

Impact on project organization and (global) development

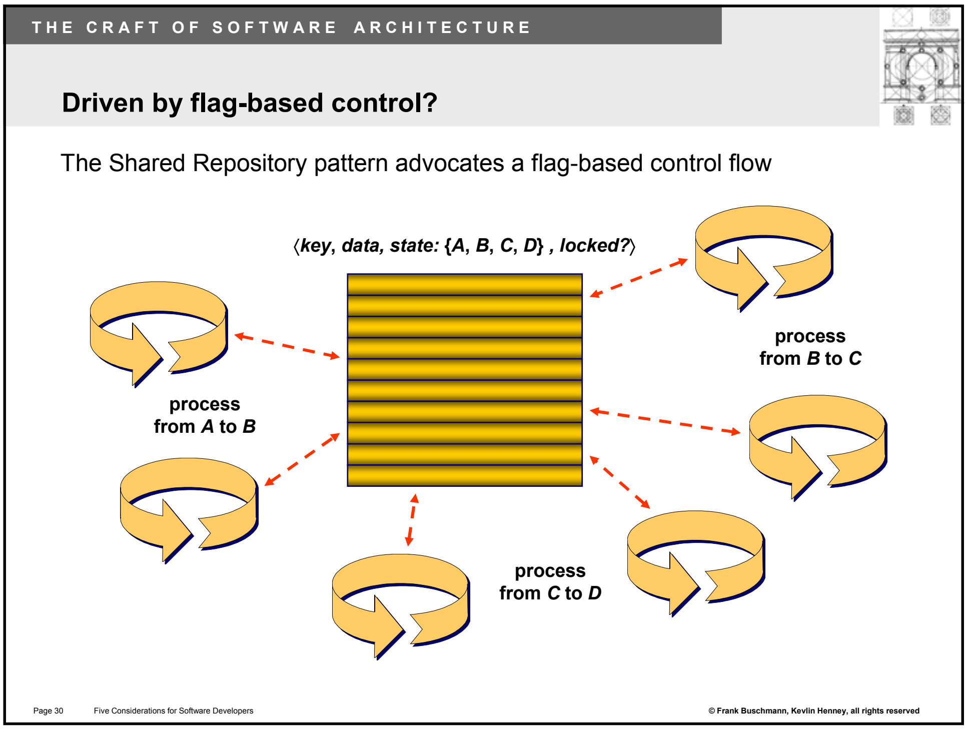
A symmetric design “created” by realizing complementary functionality using the same patterns and to the same level of detail



Considerations on design quality

Agenda

- Introduction
- Economy
- Visibility
- Spacing
- Symmetry
- **Emergence**
- Outroduction

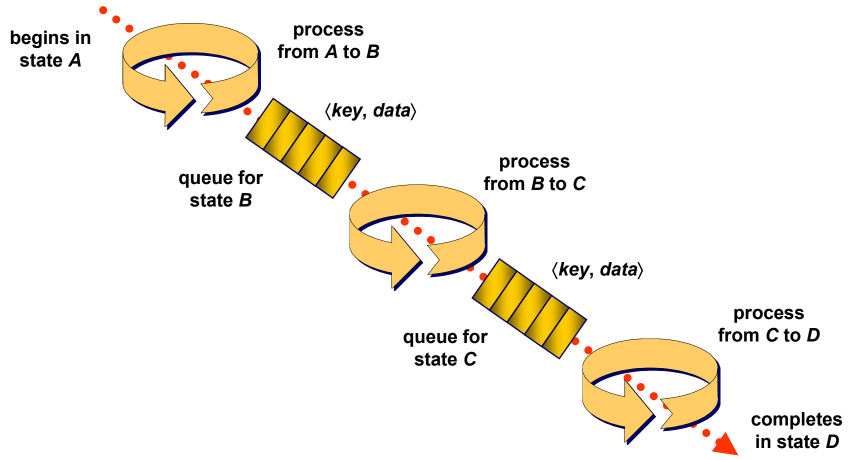


Driven by flag-based control?

The Shared Repository pattern advocates a flag-based control flow

Or driven by flow?

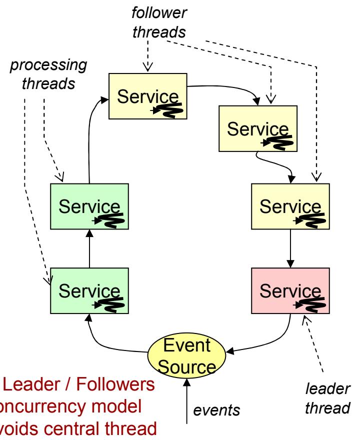
The Pipes and Filters pattern supports a data-flow-based flow of control



Command(-less) and control(-free)

Sometimes the most effective way to achieve a desired effect is to give up tight control. For example

- Self-organizing versus micro-managed teams
- Make a problem visible to encourage its solution – build problems, bug count or age – as opposed to making its solution a commanded responsibility
- Take decisions through polymorphism instead of *if*
- A sequence of elements does not need to have *sort* applied to make it sorted: start from nothing and add elements so that a sorted order is preserved



A Leader / Followers concurrency model avoids central thread management

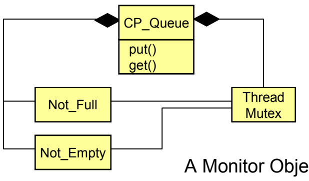
Design discovery

A RUF*-then-refine rather than a BUF** approach helps to converge on a good design

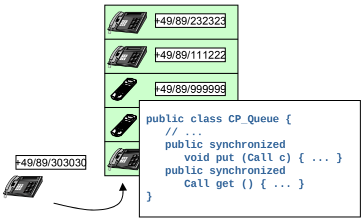
- A vision of what is needed, and a set of possible outcomes in mind, can make a big difference …
- But this is not the same as a single fixed and overarching scheme

But: emergent design is not magic, not arbitrary and not just a matter of chance!

- It needs active attention, guidance and nurturing



A Monitor Object



```
public class CP_Queue {
   // ...
   public synchronized
      void put (Call c) { ... }
   public synchronized
      Call get () { ... }
}
```

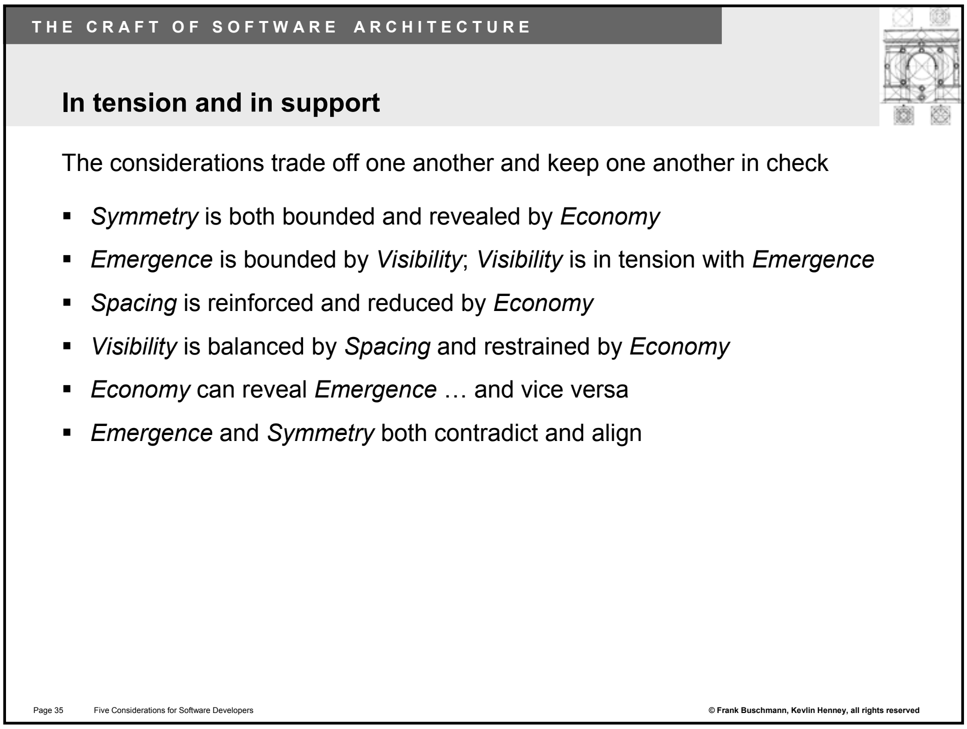
A Monitor Object concurrency model supports cooperative multi-threading

* Rough-Up Front
** Big-Up Front

Considerations on design quality

Agenda

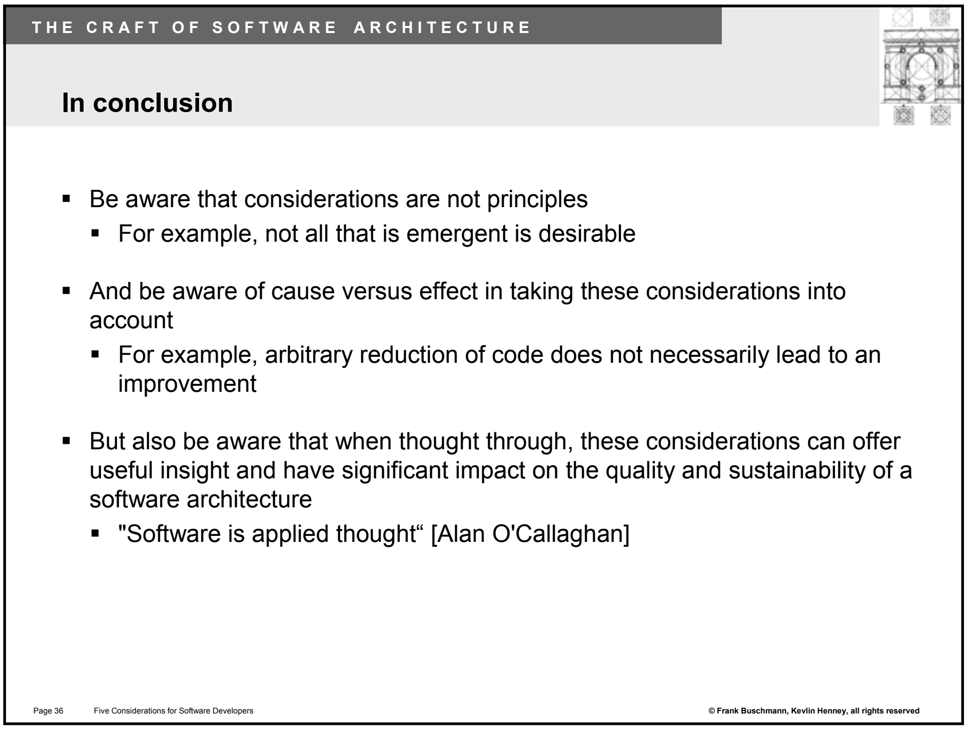
- **Introduction**
- **Economy**
- **Visibility**
- **Spacing**
- **Symmetry**
- **Emergence**
- **Outroduction**



In tension and in support

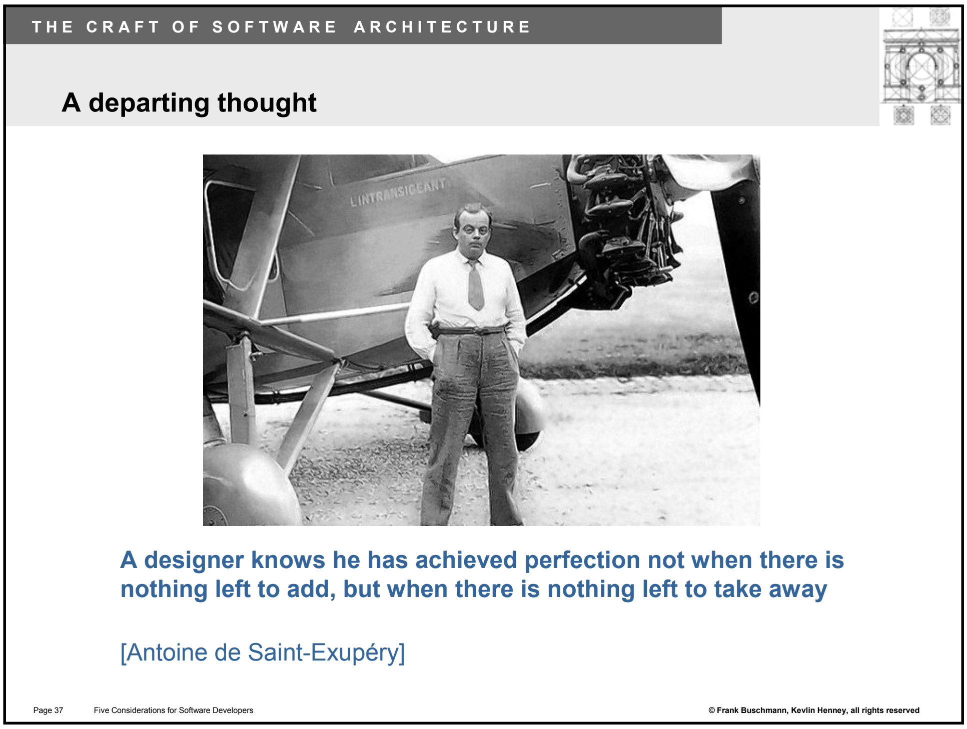
The considerations trade off one another and keep one another in check

- *Symmetry* is both bounded and revealed by *Economy*
- *Emergence* is bounded by *Visibility*; *Visibility* is in tension with *Emergence*
- *Spacing* is reinforced and reduced by *Economy*
- *Visibility* is balanced by *Spacing* and restrained by *Economy*
- *Economy* can reveal *Emergence* … and vice versa
- *Emergence* and *Symmetry* both contradict and align



In conclusion

- Be aware that considerations are not principles
  - For example, not all that is emergent is desirable
- And be aware of cause versus effect in taking these considerations into account
  - For example, arbitrary reduction of code does not necessarily lead to an improvement
- But also be aware that when thought through, these considerations can offer useful insight and have significant impact on the quality and sustainability of a software architecture
  - "Software is applied thought“ [Alan O'Callaghan]



A departing thought

L'INTRANSIGEANT

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]