

SOA Symposium

Amsterdam 2008-08-07 - 08

The Composability Index

Sven-Håkan Olsson

Why a Composability Index?

- To be able to get a fast approximation of a suggested SOA design's usefulness, when composition is expected
- Through the reasoning behind the Composability Index per se, general knowledge about real-life behaviour of SOA interfaces gets increased
- Through the work of inspecting a suggested SOA design to calculate the Index, insight increases and the design may get improved

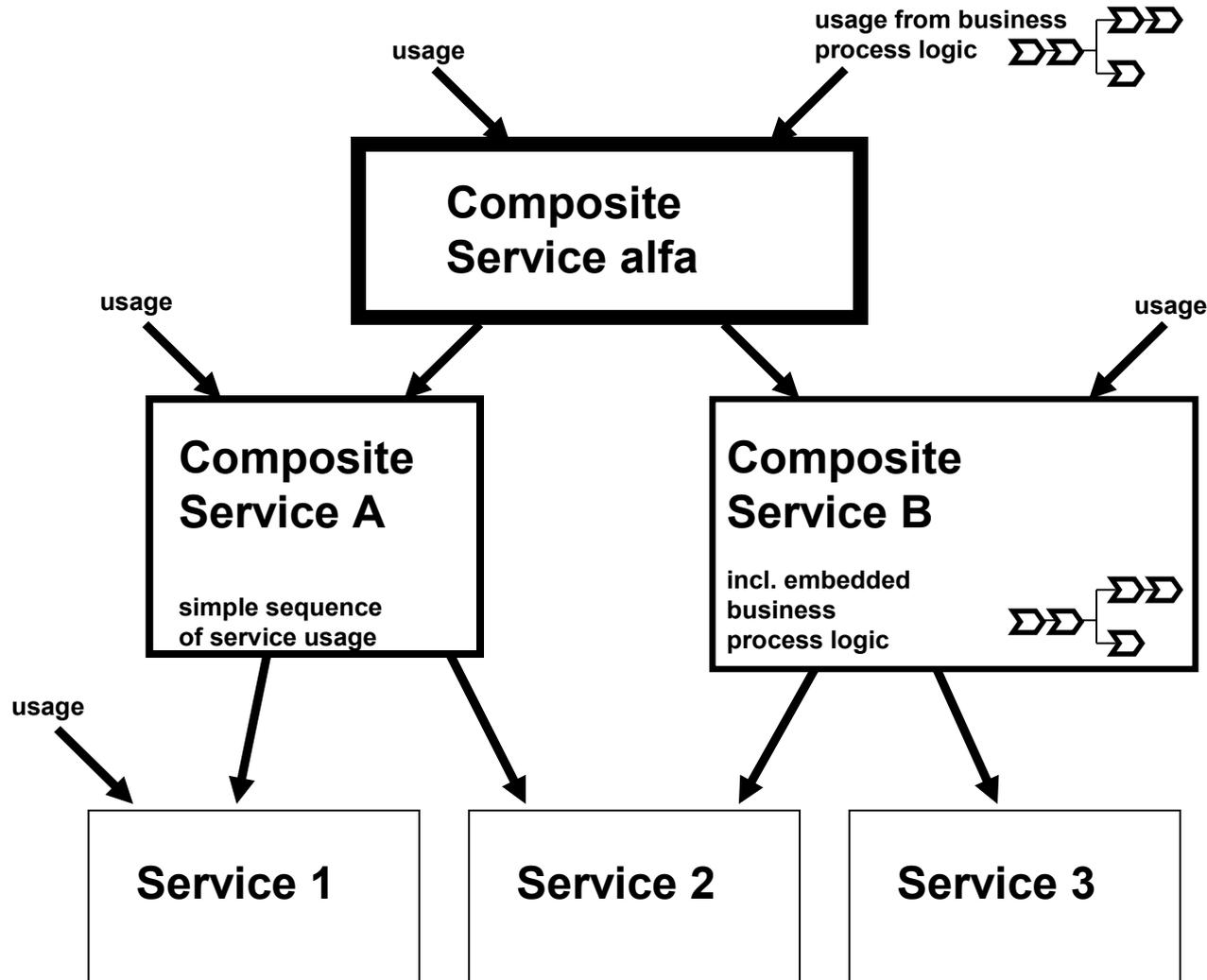
"KISS"

- The Composability Index is kept very simple so that we don't get dragged into overly theoretical metrics.
- It is a coarse approximation that should work fine for a majority of designs and usages (but not for special cases).
- "Rather roughly right than exactly wrong"

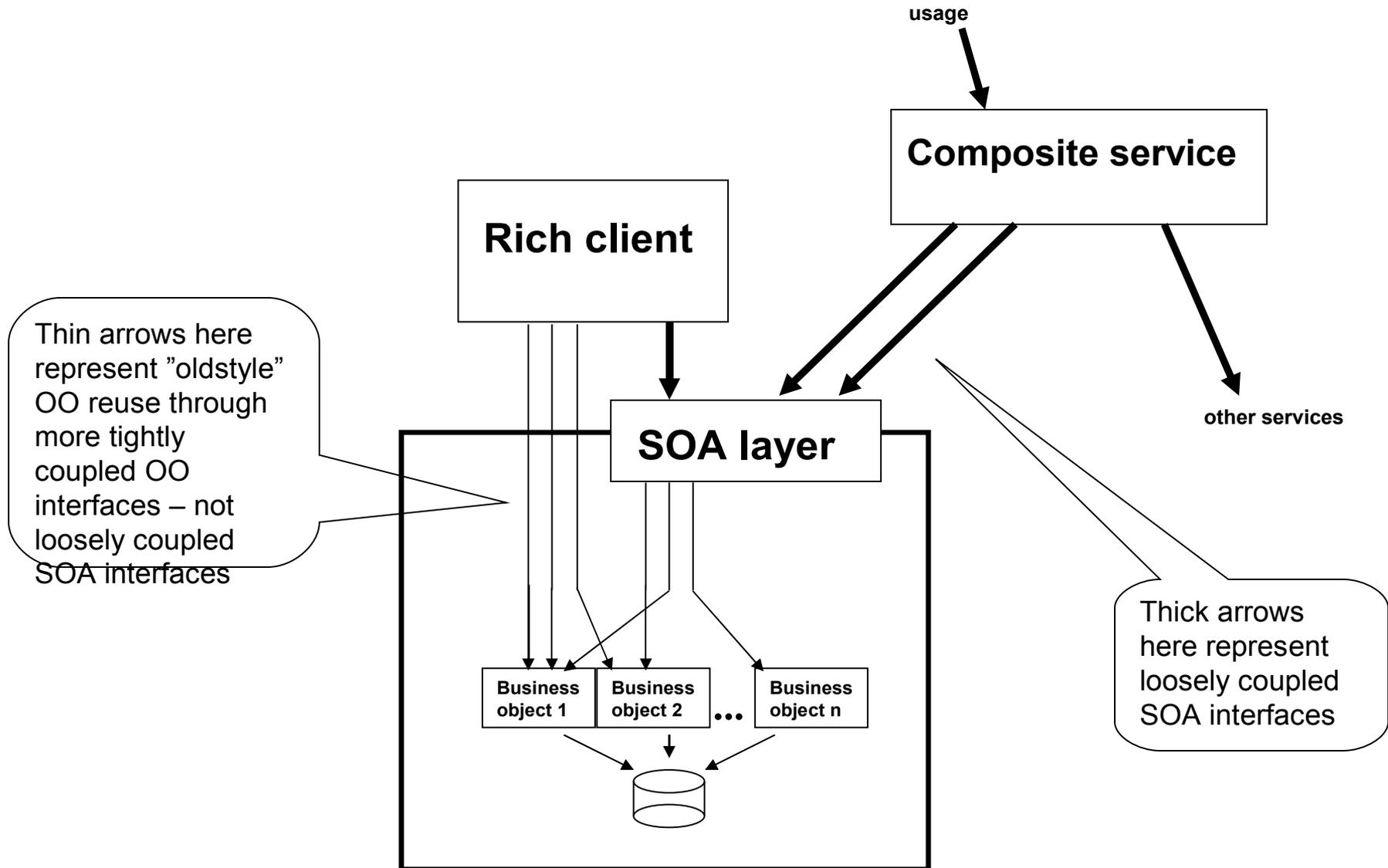
"Aren't the composability aspects included in the Index just normal SOA design criteria?"
- Well, yes, but the selected aspects are extra important for composition usages!

What does Composition mean in this session?

- To be able to aggregate several services into larger (or more specialized) services. Examples:



A more concrete example



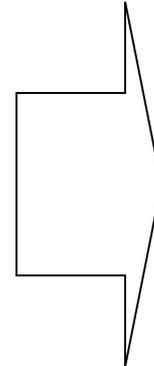
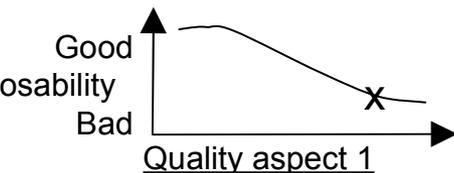
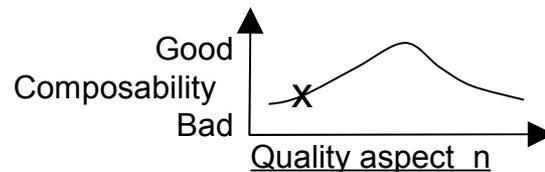
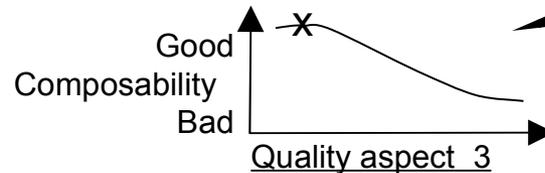
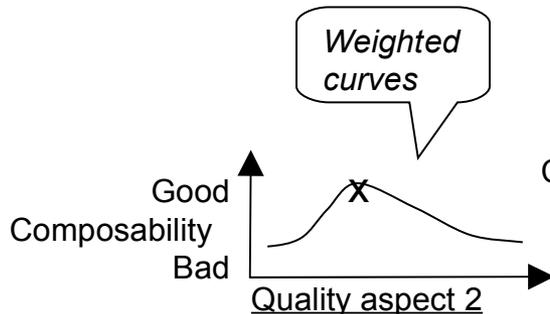
What does Composability mean in this session?

- How well it works to use a specific service interface as a part of a composition
- Mainly architectural and technical, real-life behaviour aspects
- Of course, there are other important composability aspects outside the scope of this proposed Index, such as who is responsible for what, pricing, SLA:s and many more governance aspects. Could also be included, the index is customizable.
- Also see chapter 13 in "SOA Principles of Service Design" by Thomas Erl for a thorough, less technical description of composability.

The Index

- Each service interface that is a candidate for being used in composition is inspected. A simple questionnaire is filled in with a course grading of important interface quality aspects related to composability
- The grades have weights that reflects how bad or good this is considered to be in relation to composition
- Per service interface, a resulting composability index gets calculated.

Different situations might require different weights (curves) for a quality aspect. The index is customizable!



Σ \Rightarrow

The Index

The composability quality aspects

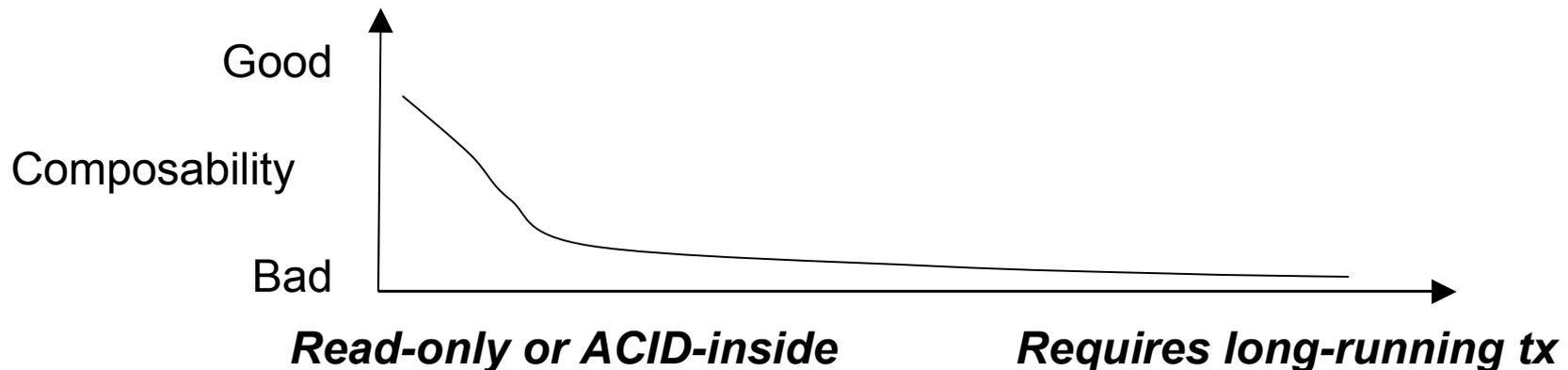
The following slides contain a number of these composability quality aspects. Included are:

1. The ACID problem for updating services
2. Loop-invocation expectancy
3. Coherence vs multi-functionality
4. Exception-handling quality
5. Availability
6. Authorization principle
7. Statelessness
8. Master Data Management (MDM) support
9. Semantic clarity
10. Canonical information model
11. Amount of business process logic inside a service

(Which aspects that are to be included in an Index can be customized, to fit different situations.)

1 The ACID problem for updating services

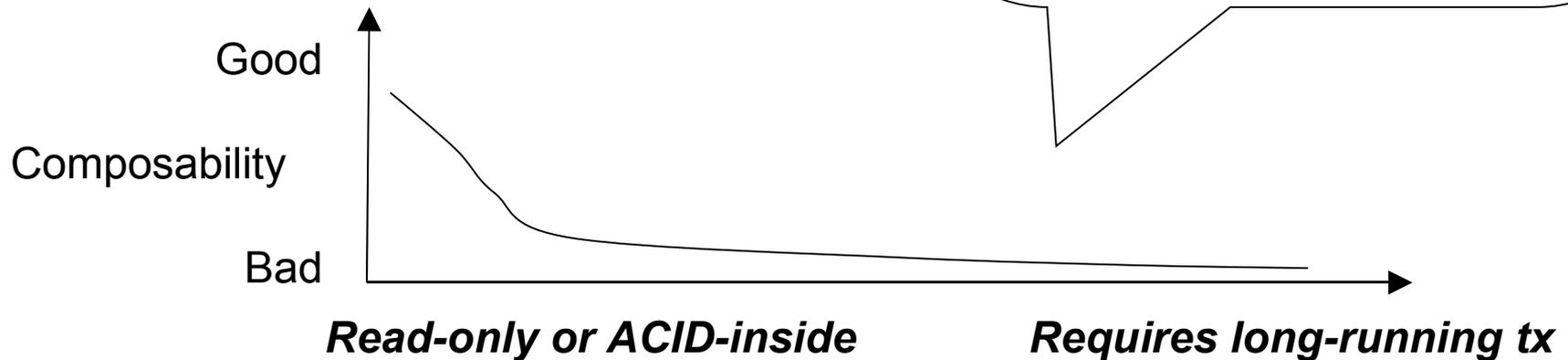
- Information fidelity and consistency cannot be guaranteed if we do not do ACID updates. Since SOA ACID normally would be too tightly coupled, we may have to take care of update errors in a couple of days (via long-running transactions).
 - Either
 - Good composability: Read-only services. Or update services that include the ACID needs inside of them.
 - Or:
 - Updating services that are likely to be used together with other updating services, thus requiring employment of long-running transactions (costly, complex, less user-friendly).



1 The ACID problem for updating services

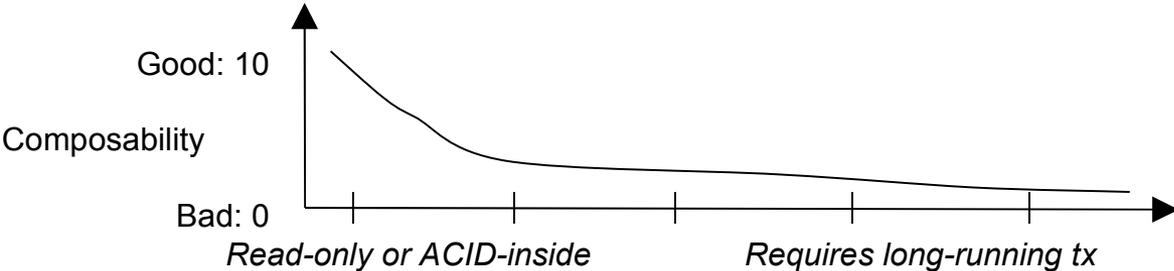
Better to make updating services that are larger, so that they can contain needed ACID handling inside the service (inside, it is OK to use tight coupling that often wouldn't be OK for SOA).
E.g.: One single SOA interface invocation for updating invoice-head together with all invoice-rows)

Long-running business transactions caused by the absence of ACID should be kept to a minimum, but cannot always be avoided.
Cumbersome because long-running transactions increase the volume of needed business logic and because personell usually have to handle the compensation.
Also, meanwhile the long-running tx is pending, the users must be informed that the info is only preliminary



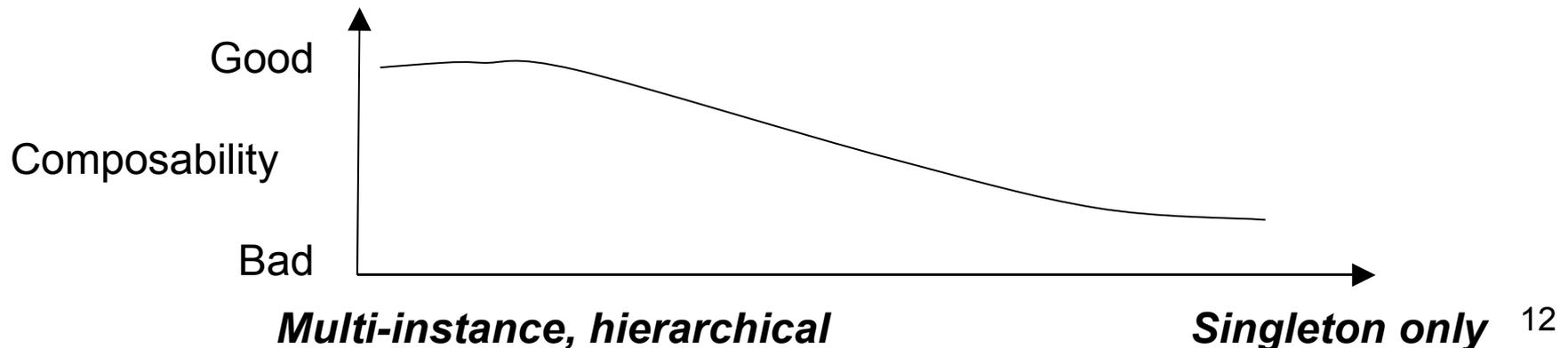
Questionnaire sample for the ACID problem

- Check one of the following:
 - This is a read-only service interface.
Or, all conceivable updates that should be kept together, are kept together inside the service, through internal ACID "Good-weight": 10
 - Internal ACID is used for related updates that should be kept together, but at rare times, related info is expected to have to be updated via another service "at the same time" "Good-weight": 4
 - Internal ACID is used for related updates that should be kept together, but sometimes, related info is expected to have to be updated via another service "at the same time" "Good-weight": 3
 - Internal ACID is used for related updates that should be kept together, but often, related info is expected to have to be updated via another service "at the same time" "Good-weight": 1
 - No internal ACID is used, several service invocations have to be carried out to complete update of related info. "Good-weight": 0



2 Loop-invocation expectancy

- Is the SOA interface designed in such a way that it is probable that the using service have to create a loop to invoke the service a lot of times, increasing overhead and latency impact substantially?
 - Either:
 - Interface that can cope with multiple instances and also with a hierarchy of instances inside of instances.
 - Or:
 - Interface that is "singleton" and only can cope with one instance of an object



2 Loop-invocation expectancy

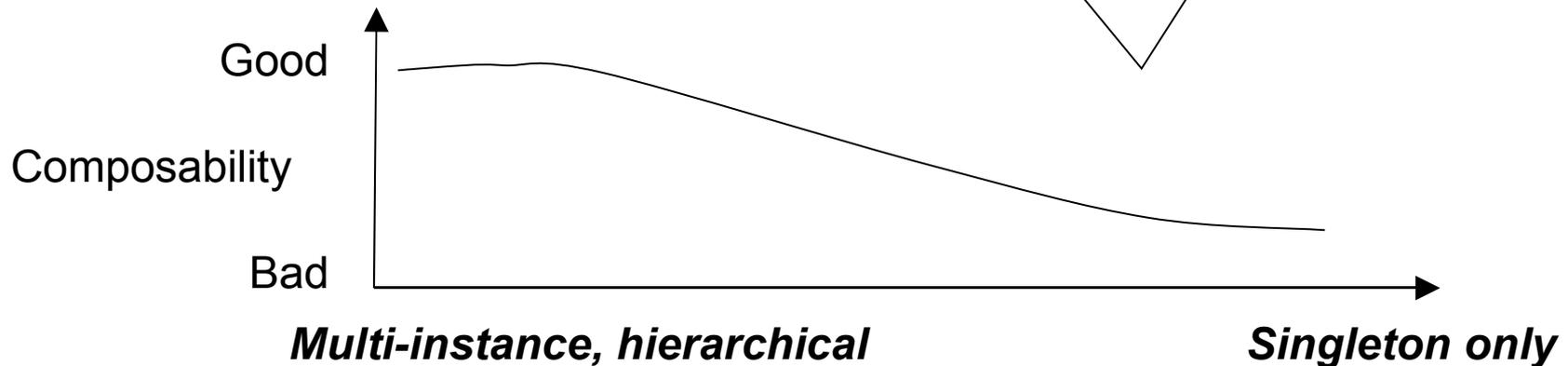
Better to make SOA interfaces that can cope with several instances of objects. Everything can be accomplished in one single invocation instead of many small invocations.

E.g.: One single SOA interface invocation for reading invoice-head together with all invoice-rows

A SOA interface that can cope with single instances can be useful, but if they are restricted to that, a large number of invocations have to be made to get complete business information.

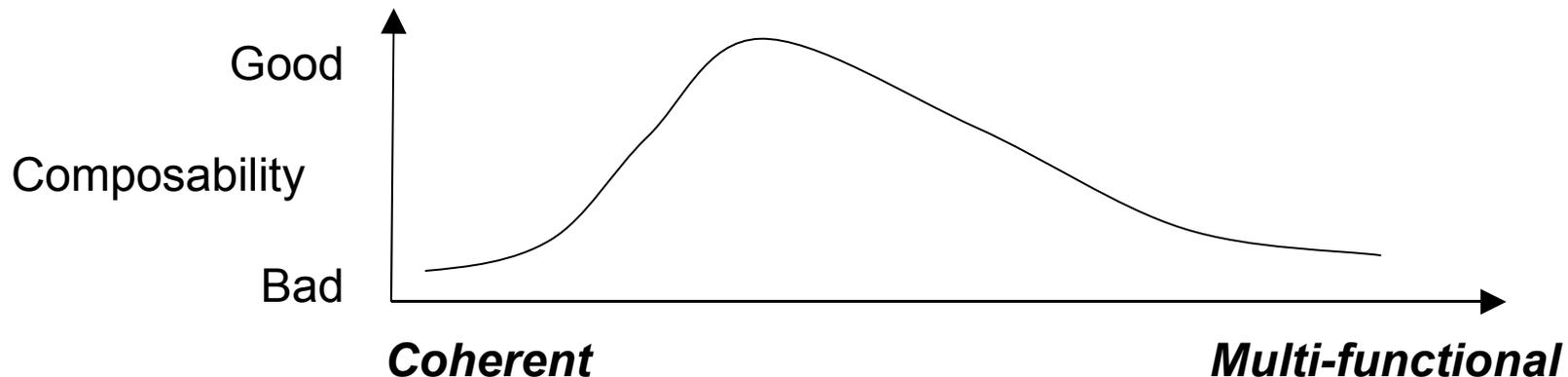
The major problem is that latency for each invocation degrades performance severely. Latency is often caused by network latency, many layers of sw, initial object creation in sw, XML-DOM creation, for satellites the speed of light etc.

Latency doesn't improve so much, whereas bandwidth does!



3 Coherence vs multi-functionality

- “How good is the Swiss army knife approach to services”?
 - Either:
 - Coherent – only one, fixed functionality per SOA interface (or just few), increases the number of interfaces a lot but each one is easier to understand
 - Or:
 - Multi-functional – complex, more abstract, reduces number of SOA interfaces but each is more difficult to understand and find

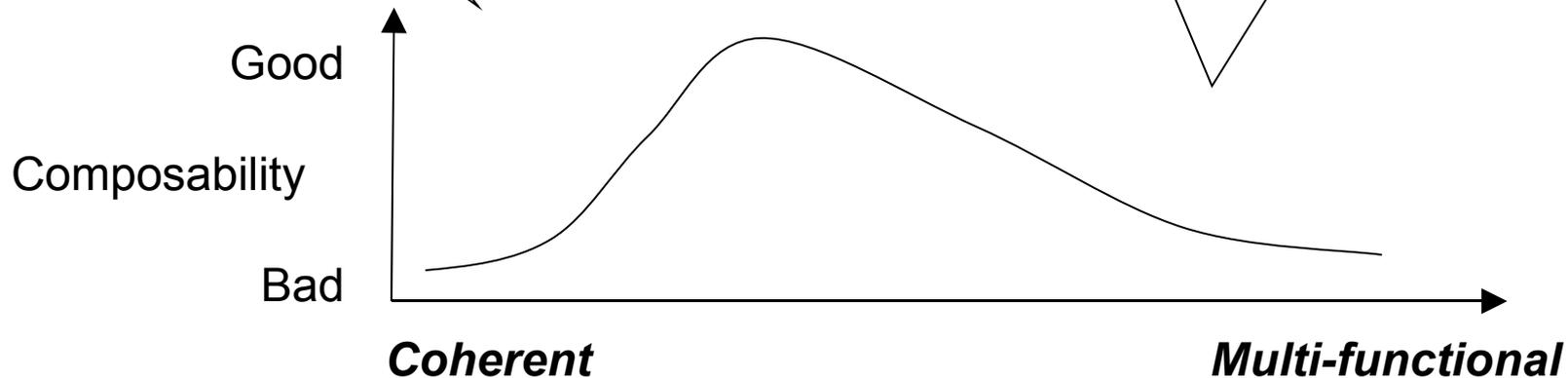


3 Coherence vs multi-functionality

Too coherent service interfaces, i.e. one interface only do exact one thing, means there will be a risk that the number of interfaces explode – all combinations of requirements permute

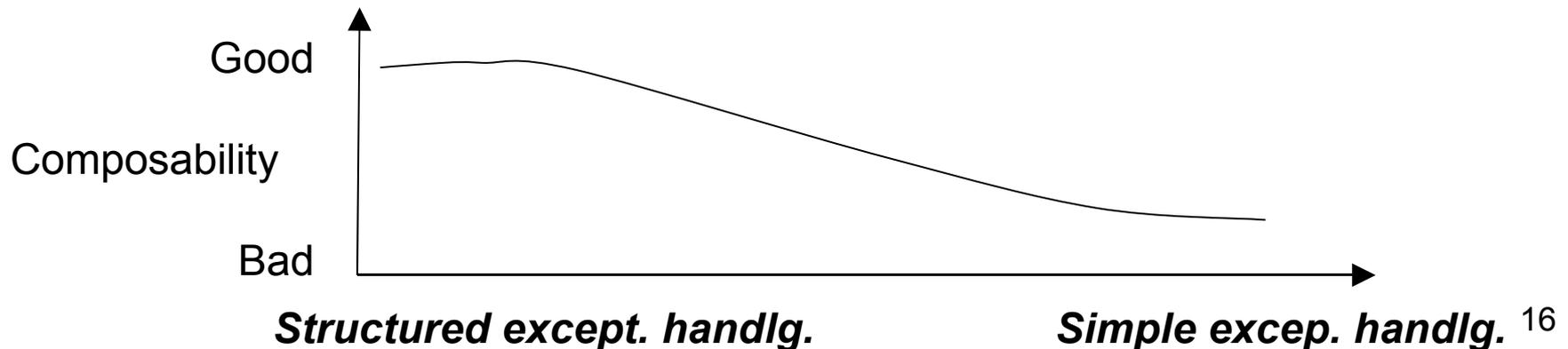
Optimal balance coherency vs multi-functionality.
Or, no multi-functionality is conceivable in this case

Too multi-funtional service interfaces will get much more difficult to understand. Discoverability decreases because the name of the interface cannot be very mnemotechnical. I.e. *AskGeneralLedger* could disguise hundreds of actual service interfaces hidden behind the façade. Interface versioning should always be clear and explicit – with multi-fyunctional it gets possible to obscure contract version breach through steering parameters that get new allowed values.



4 Exception-handling quality

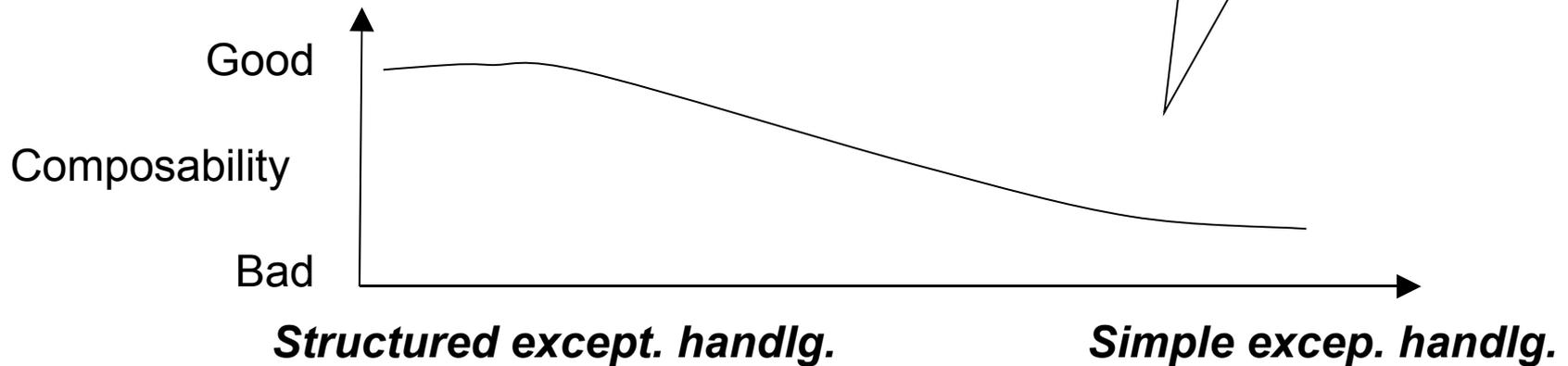
- SOA service usage must be stable and rugged, especially when combined through composition – very good exception handling is needed.
 - Either:
 - Well structured return codes. Return code descriptions. Severity levels. Possible to pass variable texts to consumer. Logging, auditing.
 - Or:
 - Overly simple exception handling – e.g. just a success/failed flag.



4 Exception-handling quality

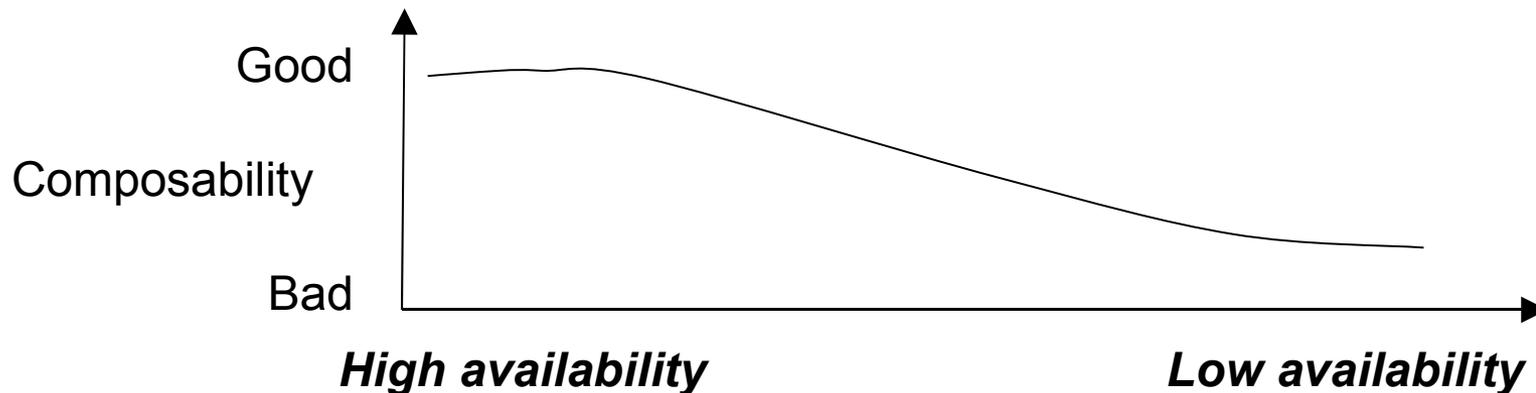
Principles for exception handling must be really well thought through. Needed SOA loose coupling means that language specifics like "throw" cannot be used. Logging and auditing should add to the possibility to find errors.

Programmers sometimes think that exception handling is boring – the result may be service interfaces that gives a less than stable total system



5 Availability

- The availability (uptime) of a composite service depends severly on the uptime of the underlying services
 - Either:
 - A used service exhibits high availability (probability figure, often part of SLA).
 - Or:
 - The service exhibits low availability



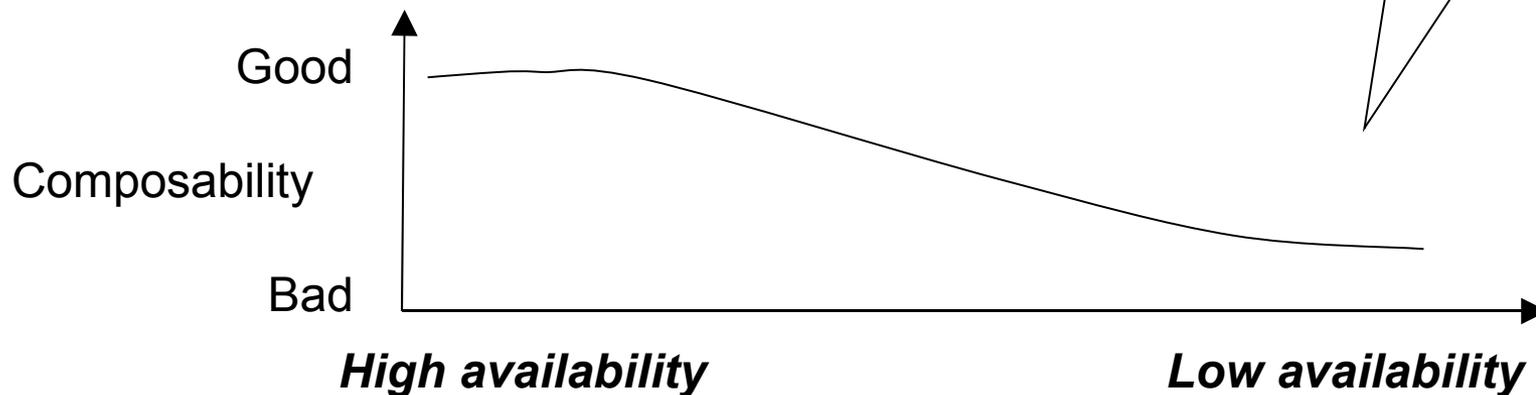
5 Availability

High availability for composed service interfaces help create a stabile total system.
The availability in turn relies on that the service's inside has fault-tolerant hw/sw, asynch nature, is well-tested & bug-free etc.

(However, cost is not included separately in this Index – high fault-tolerance can be very expensive...!)

Low availability is problematic when many services are composed together – the availability probabilities has to be multiplied.

E.g. if 10 services has 0.99 availability each, the resulting availability is only $0,99^{10} = \underline{0.90}$!

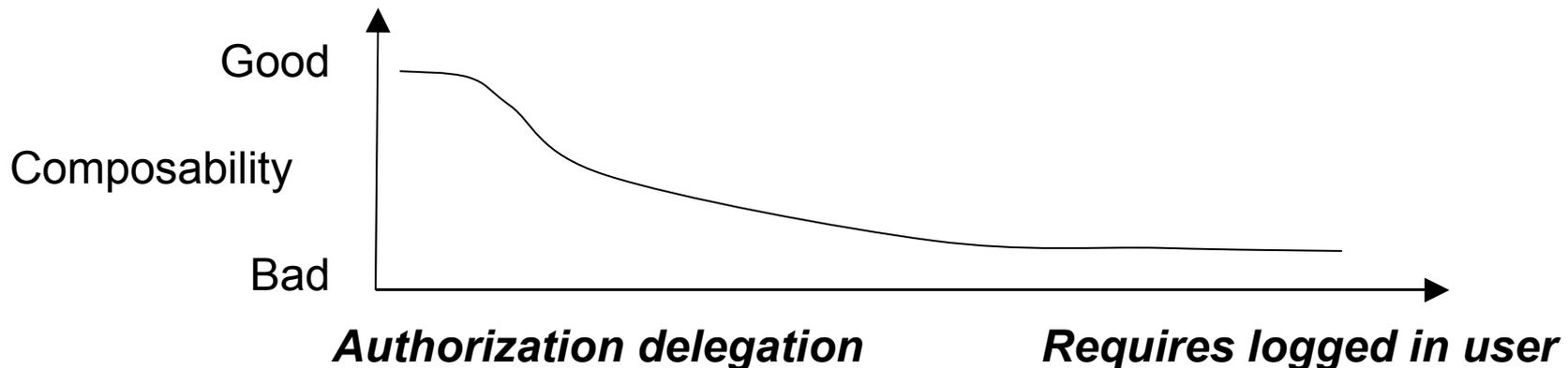


Other composability quality aspects

- The following slides include other composability quality aspects
- These are included for reference, we haven't time to go through them in this session

6 Authorization principle

- In the world of large-scale SOA the delegated authorization based on the trust principle is usually best
 - Either:
 - Authorization delegation based on the trust principle
 - Or:
 - The service itself requires to know a logged in user via some mechanism. Or, other complicated authorization mechanism.



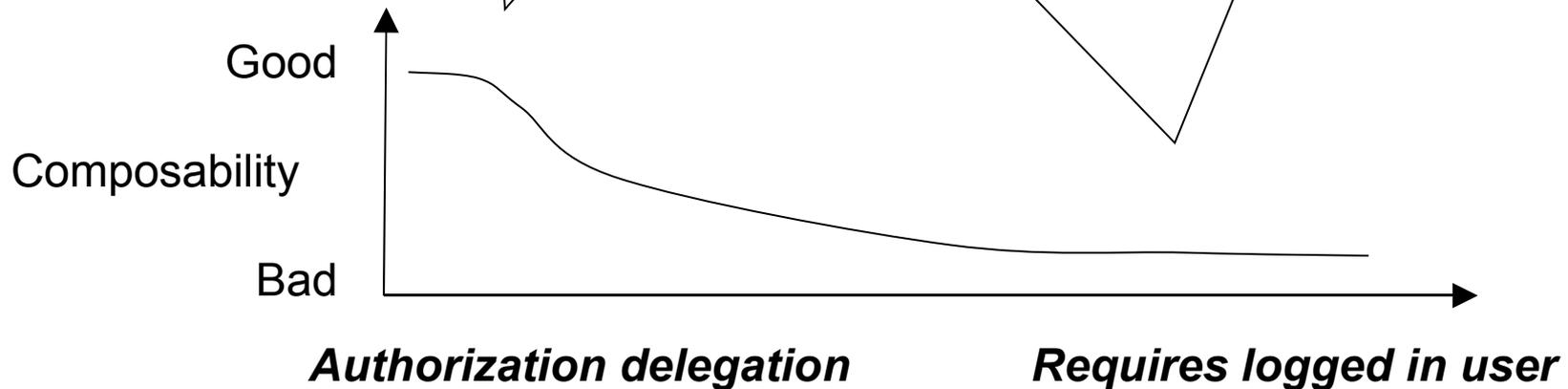
6 Authorization principle

Normally, each service should be given an authorization delegation. In this, the service carries out authorization based in its internal logic, and sometimes based on a knowledge about a possible end-user. The using services should have a trust contract with the service.

In the world of large-scale SOA it is usually not practically possible to require that all service composition layers must demand e.g. PKI log-in-credentials for a logged-in end-user.

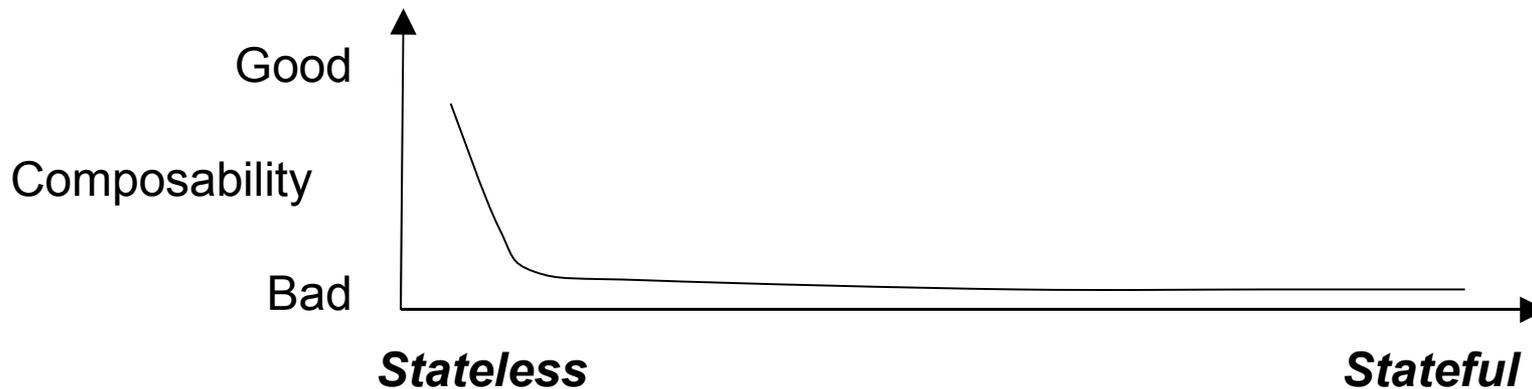
Personal PKI or other complicated mechanisms would give tighter coupling, bad interoperability and often technology lock-in. Also, sometimes there is no end-user, it's an autonomous usage.

And often, the service logic should itself have higher rights than the user per se, so simple "impersonation" would not work..



7 Statelessness

- Is the service interface stateless?
 - Either:
 - It is stateless, so that it doesn't rely on other service invocations in a specified sequence
 - Or:
 - It is stateful



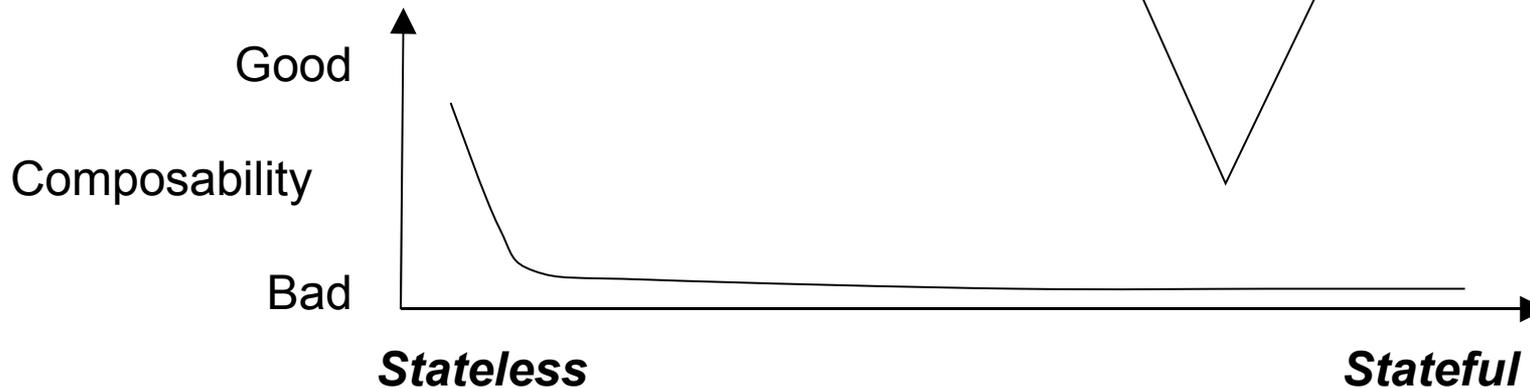
7 Statelessness

The invocation does not have to be a part of a specified sequence, that would give a too tight coupling.
E.g. a course granular interface with one invoice-head together with all its invoice-rows would be stateless.

One might argue that persistence inside a service (storing in a database) is a form of statefulness, but that is not what we focus on here, it is rather the interface's statefulness.

A way of avoiding statefulness in another way is to create the business logic so that it allows for incomplete preliminary data, e.g. then it might be OK to store orphan invoice-rows without any invoice-head.

Another statefulness that would be OK is security infrastructure state, e.g. a VPN session.

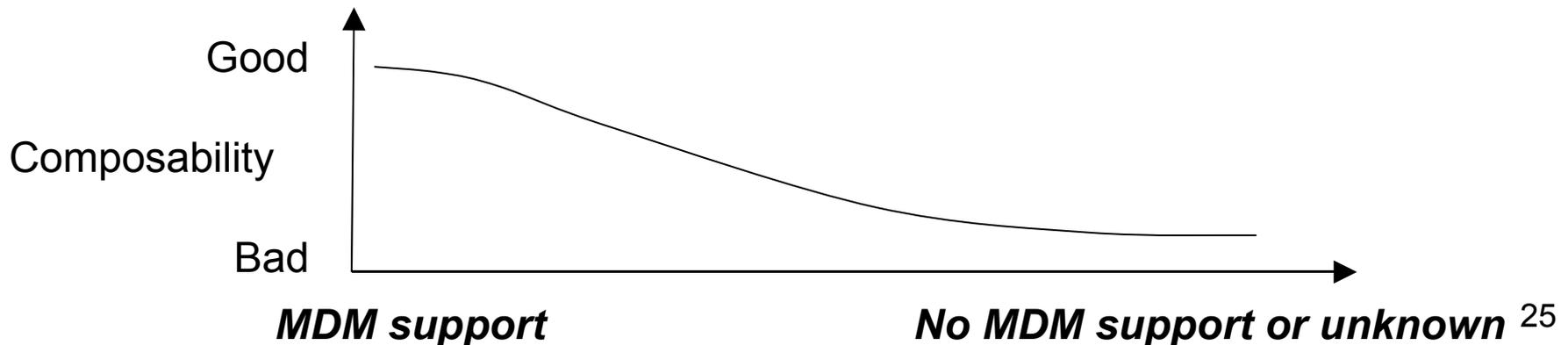


8 Master Data Management (MDM) support

- In cases where the same info is stored inside several systems (too common), a composer must have a chance to know if the service takes care of notifying other systems about updates, in an MDM fashion, or not.

Part of the contract.

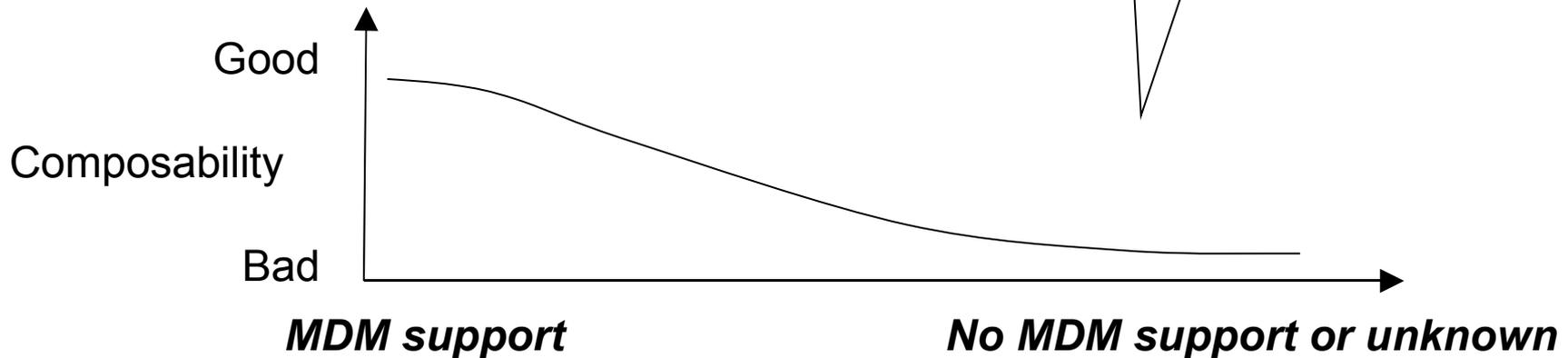
- Either:
 - The service contract states that the service takes responsibility to notify according to an MDM scheme
- Or:
 - The service contract doesn't states anything about this issue, or states that it doesn't take MDM responsibility



8 Master Data Management (MDM) support

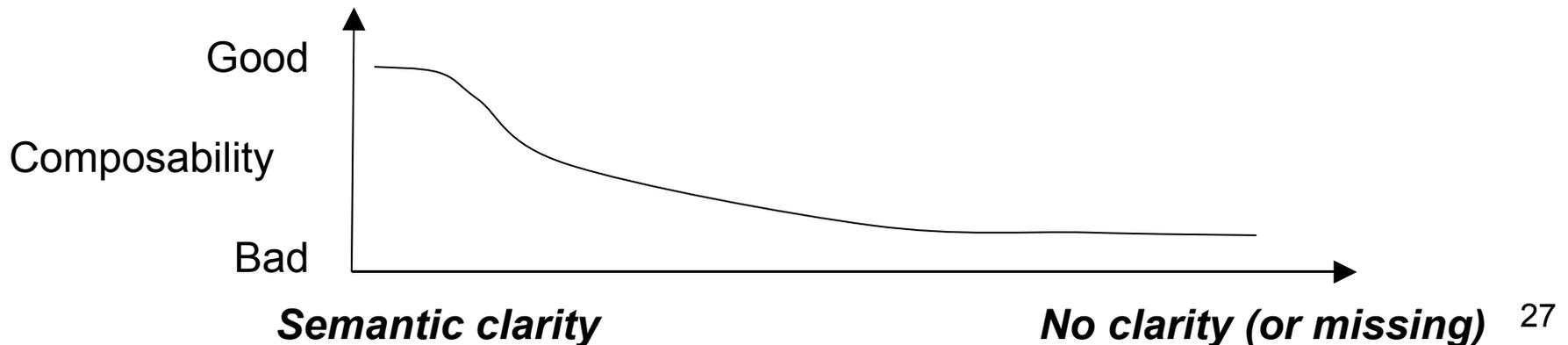
Composition is powerful, but there is also a risk that it hides problems, one of them MDM issues.
The service contract should thus tell that it supports MDM.
(In an ideal world, all MDM needs would already be taken care of behind the SOA interface, but the world is not ideal...)

The unknown state is of course not good.
And if it is stated that MDM is not included, the composite service probably have to device its own MDM, e.g. an asynchronous notification to another system.



9 Semantic clarity

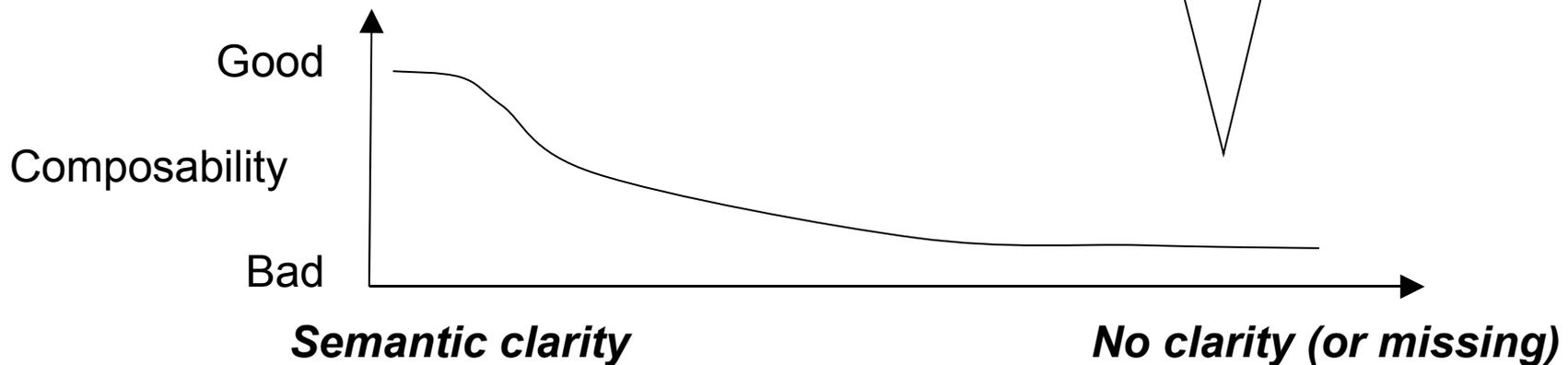
- This is a test of if the service interface contract is clear enough, specifically regarding semantic description quality.
 - Either:
 - The service interface contract contains (or refers to) a clear semantic description of its information
 - Or:
 - The semantic description is not clear (or missing)



9 Semantic clarity

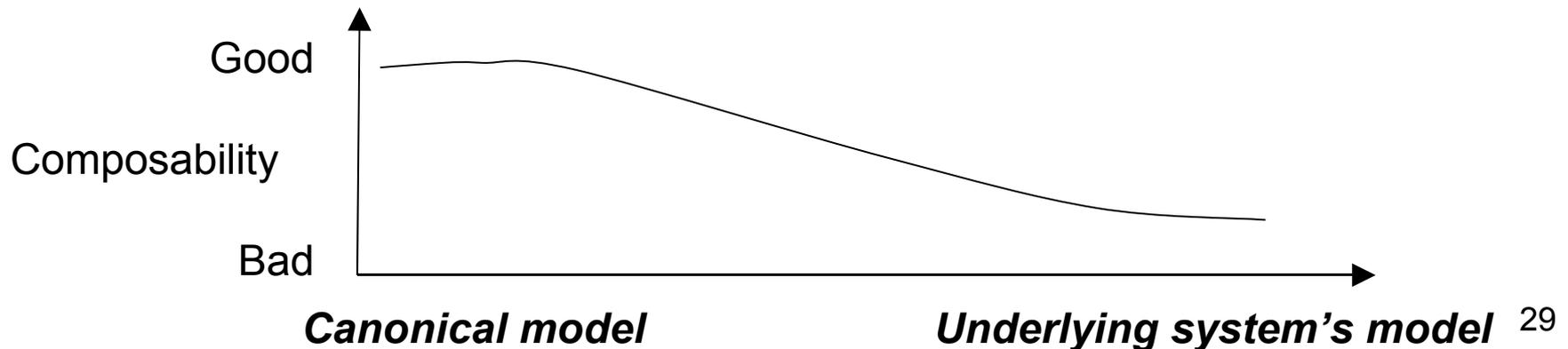
Semantic descriptions are extremely important e.g. for a composer to understand if he/she is mixing incompatible data

Often the only description of a service interface is a WSDL file and perhaps some uncommented XML schemas. That is not enough to really describe the semantics.



10 Canonical information model

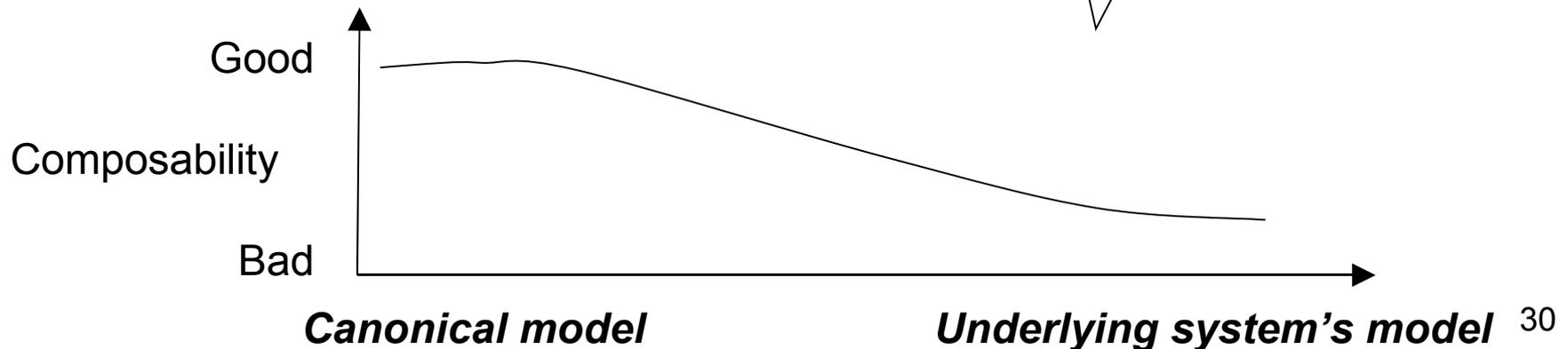
- In large-scale SOA it is probably impossible to have the same information model and the same semantic concepts everywhere, throughout the whole organization. However, if a service interface follows a common, canonical information model used for integration and SOA purposes, it should be useful when composing.
 - Either:
 - The service interface follows a canonical information model.
 - Or:
 - The information model of the underlying system is instead exposed in the service interface.



10 Canonical information model

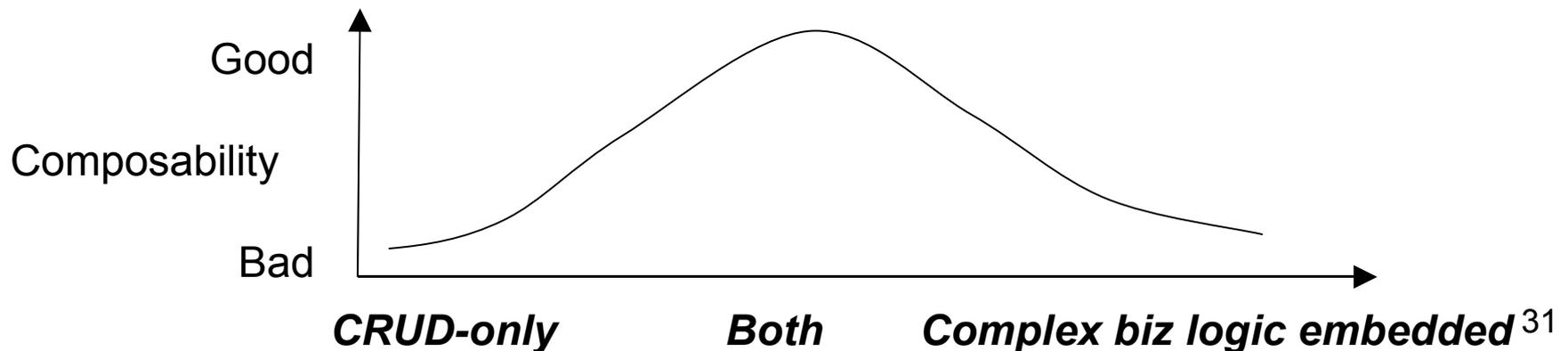
A canonical model is in theory very positive.
However, it is debated how feasible it is to define one, and to use it, in real-life.
Thus, the "good-weight" of this might be controversial – different organisations might choose to take different decisions.

One of SOA's great features is the isolation of underlying details (black-box principle).
Thus, different underlying systems may have different information models.
In the service usage (especially in composition) it is therefore sometimes to be expected to do translations between information models – or that this takes place in an intermediary layer.



11 Amount of business process logic inside a service

- Business process logic tends to need to change more frequently than the underlying information model which in turn could get exposed via simpler CRUD interfaces
 - Either:
 - Almost no business logic inside the service, e.g. only CRUD interfaces for information objects.
 - Or:
 - Complex business logic chunks embedded behind the SOA interfaces



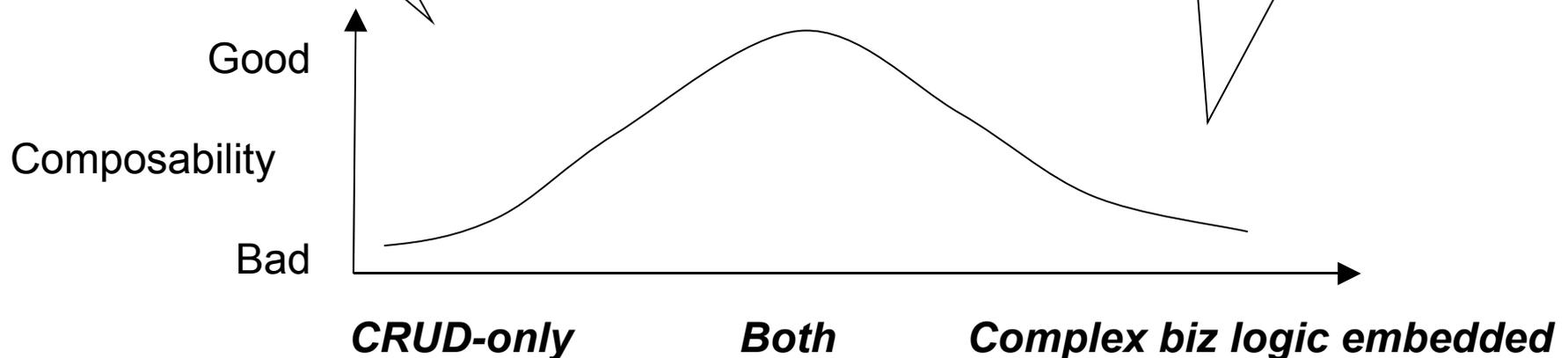
11 Amount of business process logic inside a service

CRUD-only is overly simple, in effect only a database encapsulation layer.

Composite services would not be allowed and there is a sound reuse potential also for these.

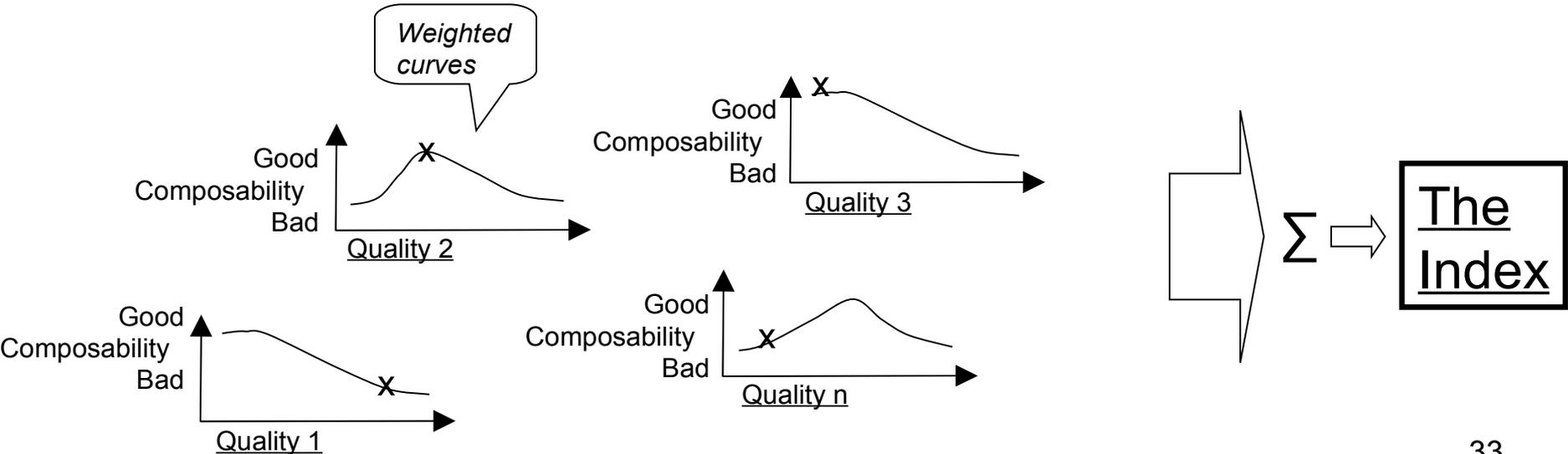
Optimal balance – there is a need both for basic CRUD interfaces and more complex logic services. Composition in itself can yield both levels in parallel.

Service interfaces shouldn't have to change every month. In an agile business some business logic will have to change frequently – in this case don't let this be handled inside services but instead in a biz process layer on top of all services – and closer to the GUI:s.



Calculating The Index for a service interface

- Very simple, just add the "score" from each questionnaire aspect (i.e. the selected grades' "good-weight").
- Divide by number of aspects. Voila, the Index!
- Easy to create an Excel/Calc sheet for the questionnaire and the calculation.

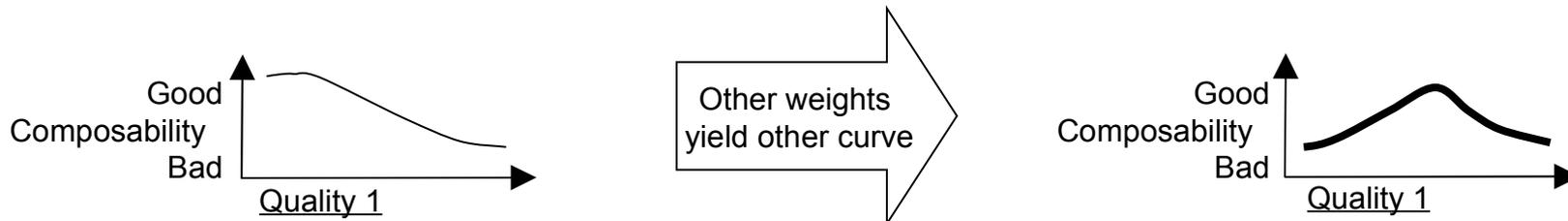


Sample calculation

<u>Service interface A's composability Index:</u>					
Composability quality aspects					Selected grade's "good-weight"
The ACID problem for updating services					2
Loop-invocation expectancy					10
Coherence vs multi-functionality					4
Exception-handling quality					2
Availability					10
Authorization principle					10
Statelessness					10
Master Data Management (MDM) support					5
Semantic clarity					3
Canonical information model					4
Amount of business process logic inside a service					5
Sum:					65
Divide by number of aspects, which is:					11
Yields the <u>Composability Index</u> for A:					<u>5,9</u>
(Weights should be from 0 to 10 in this example.)					

Customize and modify the Index?

- A global index where all the world's services could be justly compared is not within the scope of this simple metric.
- Instead, it is altogether possible to modify the index to match different circumstances in a specific organisation that wants to use composition
- Two easy ways to modify the index:
 - Which quality aspects should be included?
 - Maybe one of them is not of interest in a specific case?
Maybe another should be added?
 - Which weights should be applied on each aspect?
 - This would change the "good/bad-curve".
For example, if it is sure that the whole solution would only be deployed in one single building, then network latency may be ruled out as a problem cause (whereas in a system deployed around the world, that would definitely be a problem) – so the "Loop-invocation expectancy" aspect could be given more optimistic weights.



Questionnaire sample

- A sample Excel file with a questionnaire and calculation is available for free download at www.definitivus.se/soasymposium
- Includes
 - Short text describing each answer alternative
 - Suggested weights for the grades

Sven-Håkan Olsson is an independent consultant, course leader and speaker who focuses on application architecture and SOA. Since 1977 he has worked in a large number of IT development projects, ranging from embedded microcontrollers to main-frames. He has carried out modeling, architecture design and programming in diverse business areas. He has also specialized in reviews of problem projects.

Sven-Håkan Olsson holds an MScEE from the Royal Institute of Technology in Stockholm. In May 2008, he was appointed one of the 'top developers in Sweden' by the magazine IDG Computer Sweden.

Please feel free to contact
me if you have comments
or suggestions!

D·E·F·I·N·I·T·I·V·U·S

Tel +46 708 840134
sven-hakan.olsson[.]definitivus.se
www.definitivus.se