Component Based Production of Software

An architectural choice

HvLeunen

Headlines

- Exploits the synergy of mixing open and closed technologies
- Enables a new open market
- Solves the upcoming HR dilemma
- Effectively hides IP
- Enables robust products
- Gives all parties an equal chance

Aim

- Establish an open market for packages of components
- The packages will contain software components
- The packages may also contain hardware components or IP-blocks
- The components in the package together constitute a coherent service

Strategy

The synergy obtained by allowing an integrated design and build environment to interact with a series of publicly accessible repositories is used to establish the ultimate in reuse and in manageability of the software generation process

Consequence

- **Without programming, an architect can construct a working and testable prototype**
  - **That incorporates a tailored supporting infrastructure**
  - **That uses available components**
  - **That uses skeleton components that are designed by the architect**
  - **That uses skeleton components, which are retrieved from one or more repositories**

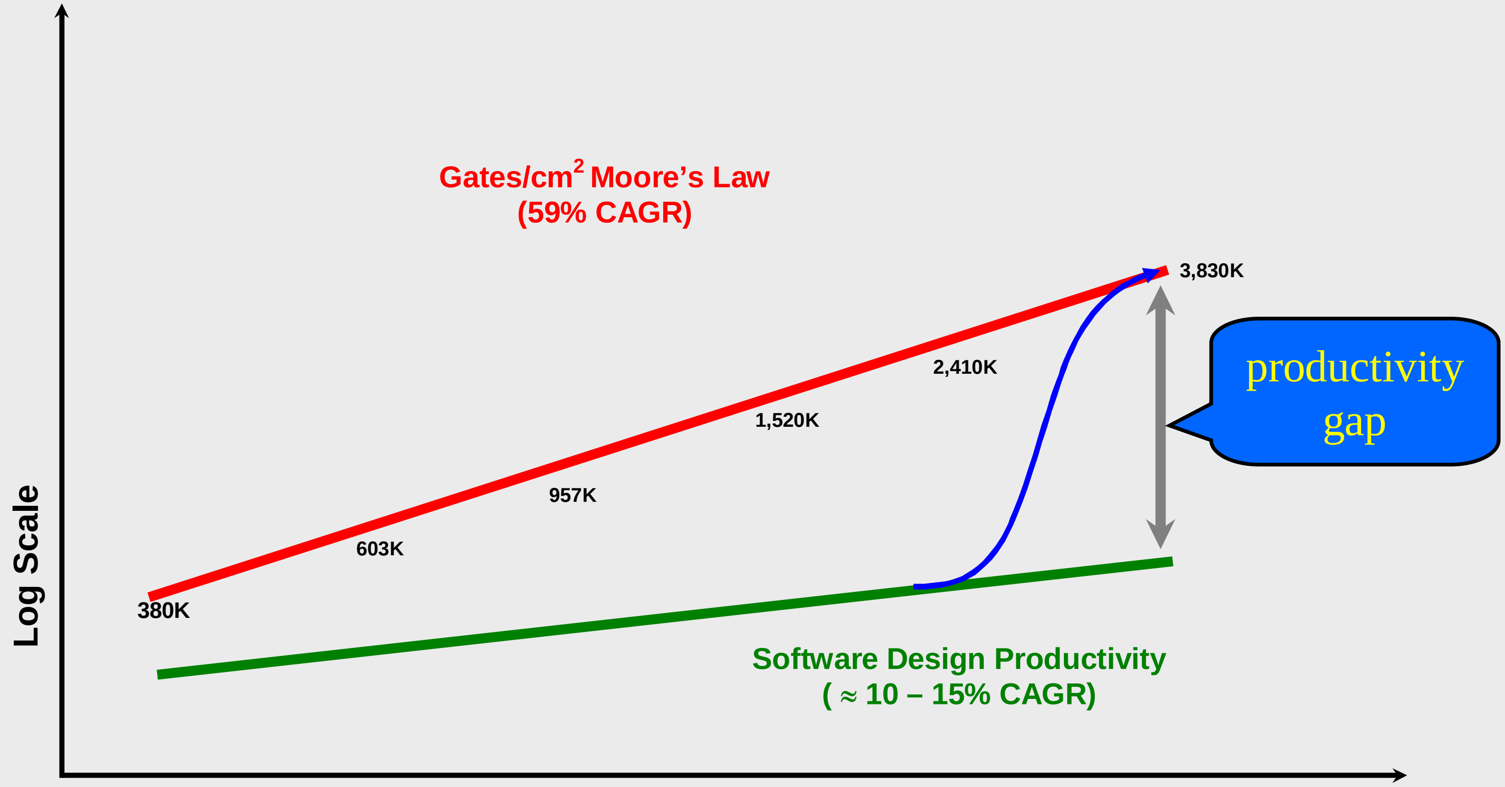
Stepwise development

- **Starting from an early prototype the product evolves by**
  - ✓ **Filling the designed skeletons with active code**
    - ❖ **This is done by domain expert programmers**
  - ✓ **Replacing retrieved skeleton components by active equivalents that are obtained, e.g. via e-commerce transactions with the site where the corresponding repository is located**

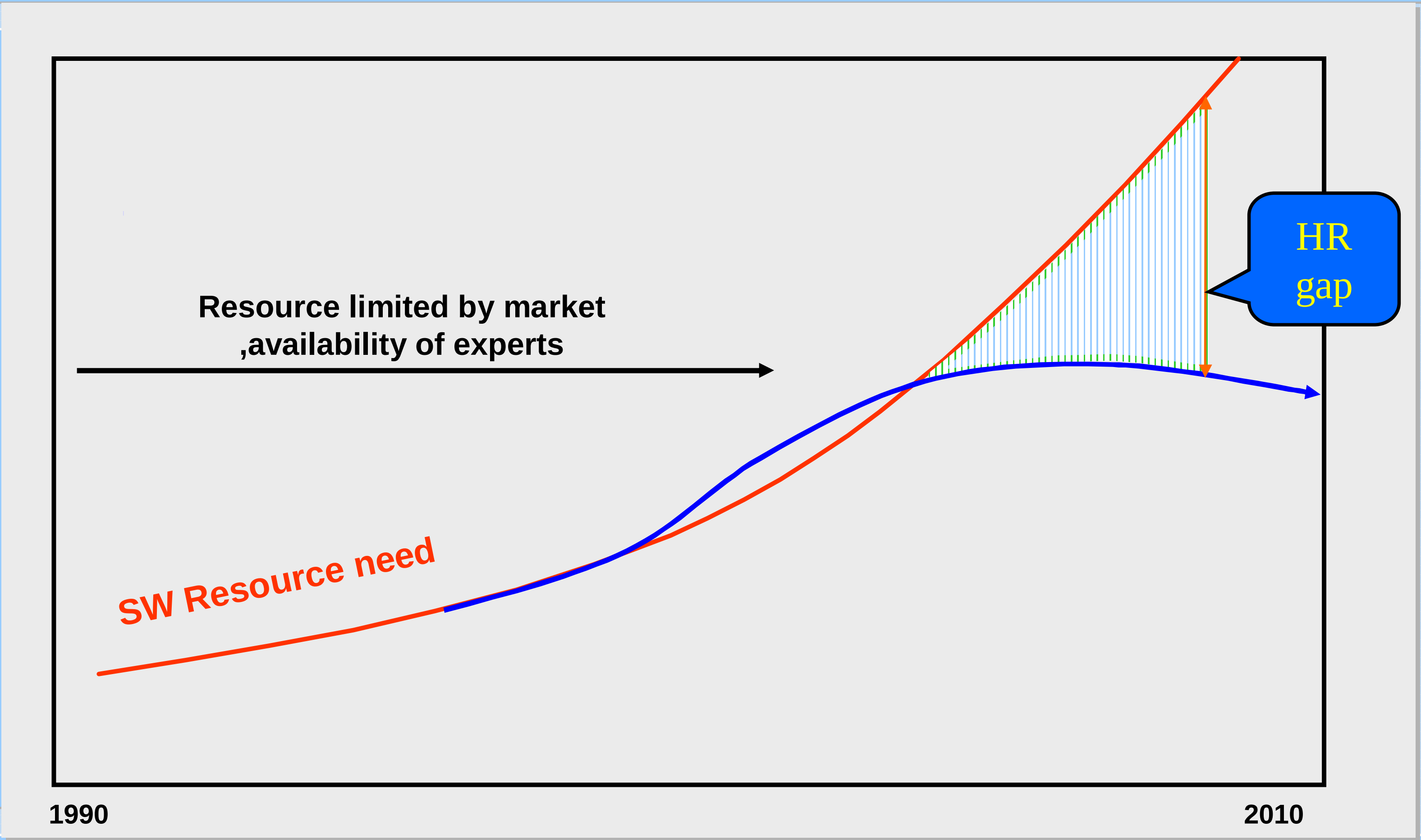
Why this strategy

The market pressure

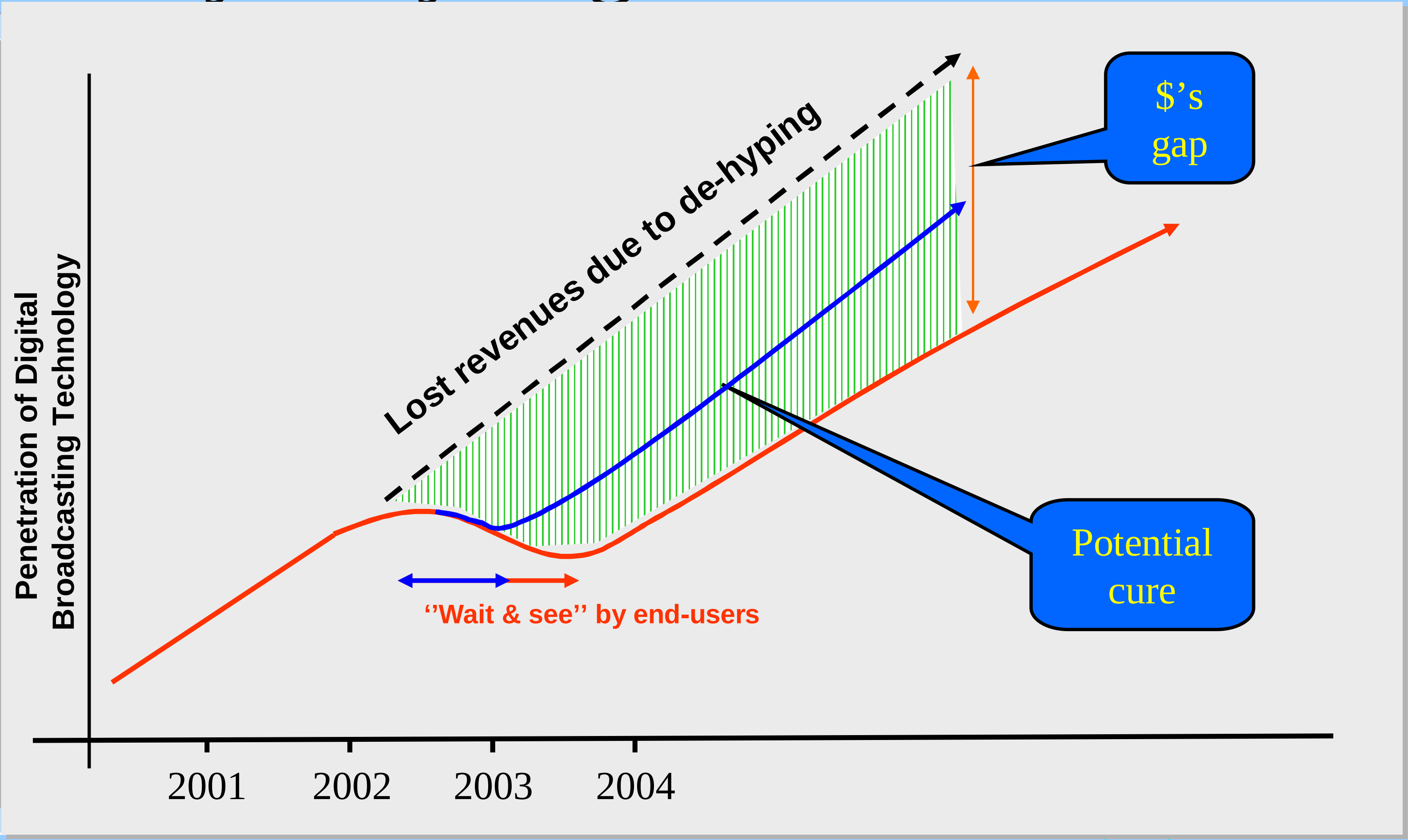
Design Productivity Gap



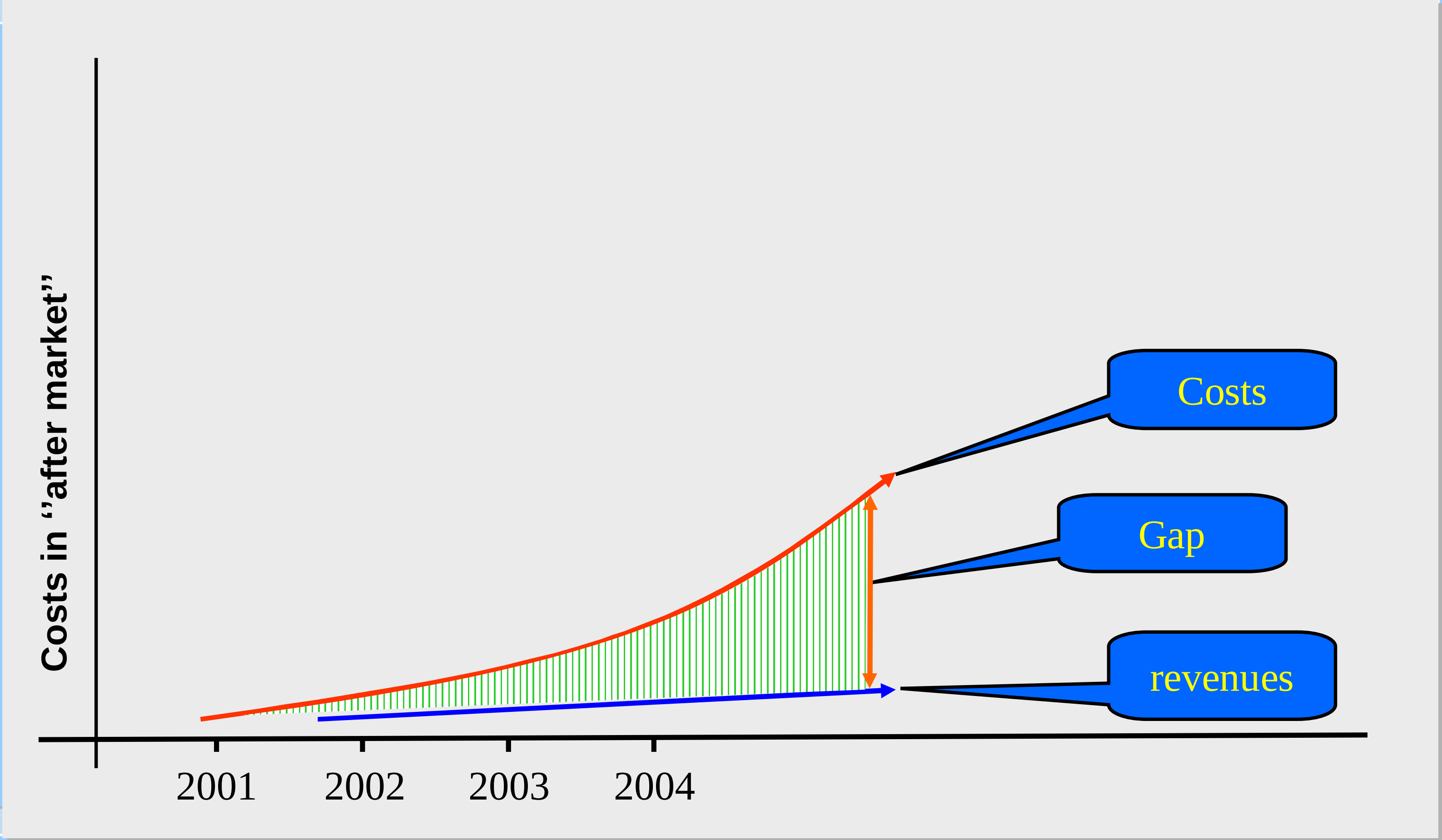
Increasing Hiring + Cost dilemma



Growth in digital market delayed by bugs



Recall and download costs



Why this strategy ?

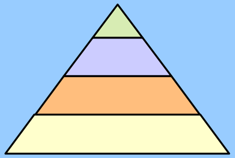
The more technical reasons

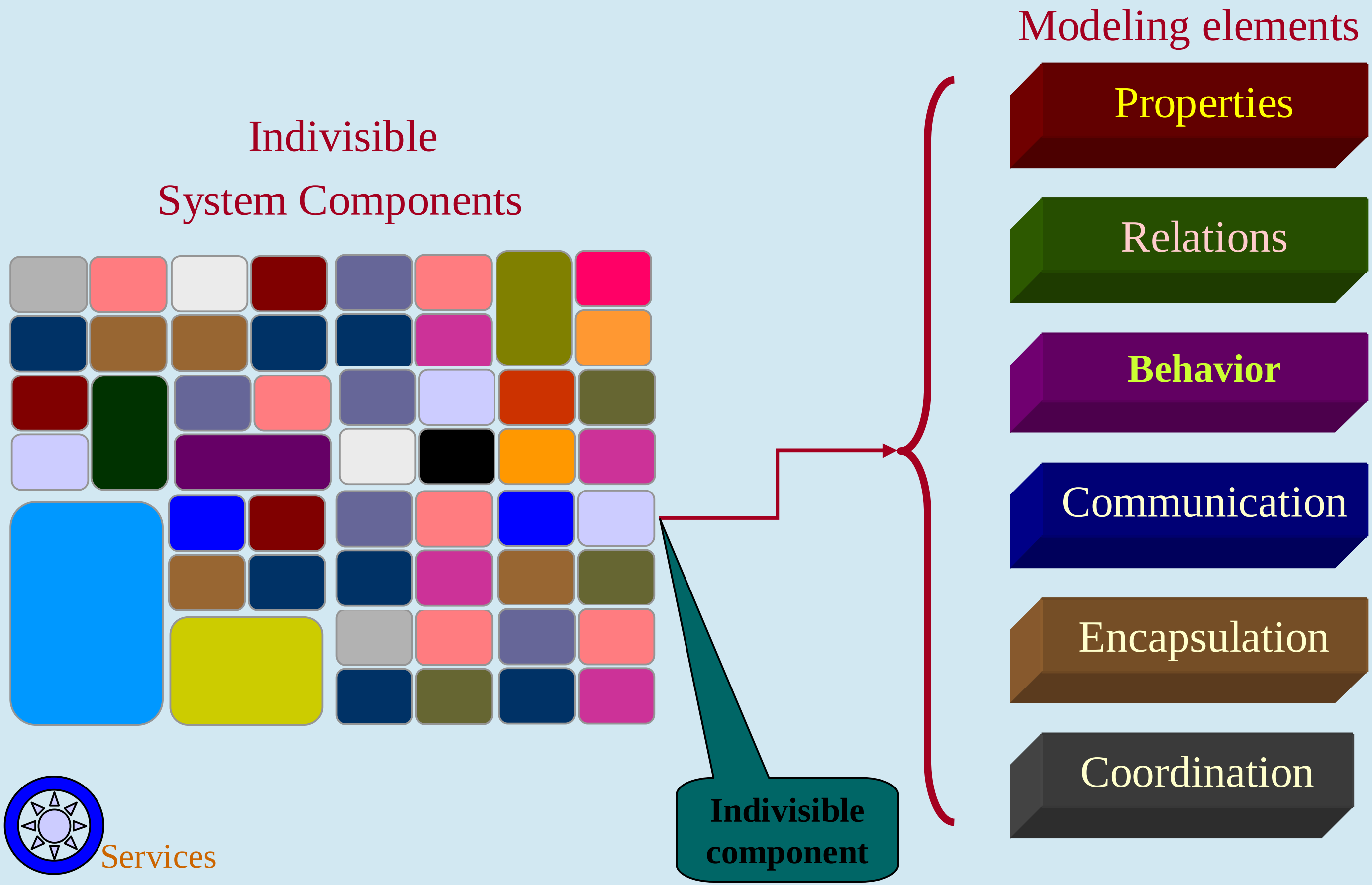
- Manageability orders of magnitude better than conventional ways
  - r·n·(n-1) rule
- Optimal reusability
  - Well defined modules
- Secure IP hiding + publishable interfacing
  - Architecture bipartition
- Optimal configurability
  - Lego-like & tool supported
- Vast reduction of time to market
- Vastly reduced project risk

The E = m ·c^2 of component technology

Explicit exposure of metadata & meta-models

Modularization





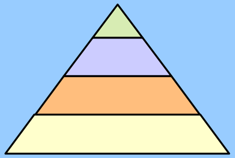
Speaker Notes

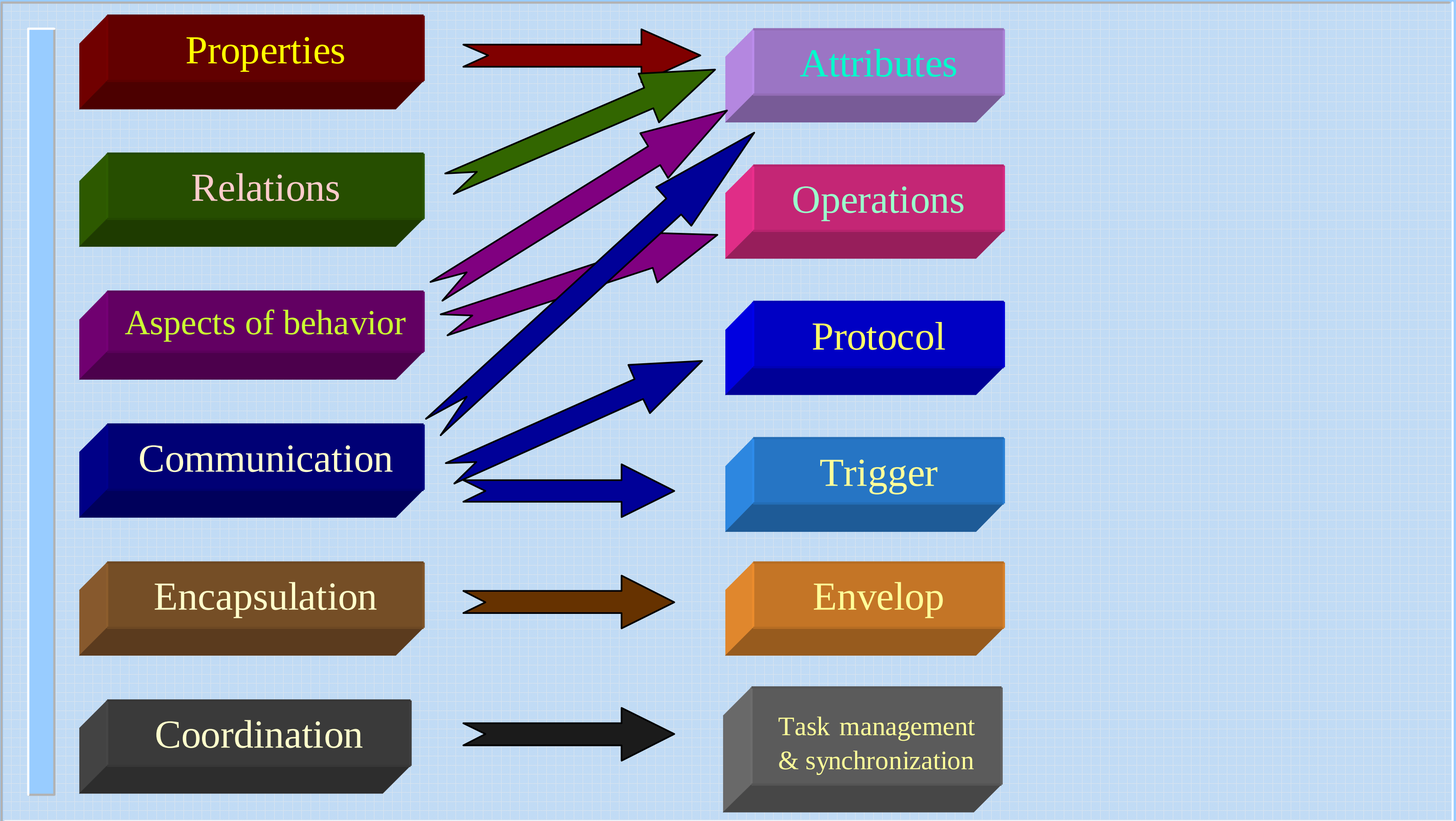
A system can be componentized by splitting it into a series of components that are as independent from each other as is feasible. In this process the central services are set aside as a special category. On their turn these components can be split in smaller components. This process continues until the components can no longer be split into smaller components. Still these components can be subdivided into a series of modeling elements. These are:

properties, relations, aspects of behavior, communication, encapsulation and coordination.

Coordination takes place between components and between components and the central services of the system

Modeling Elements





Properties

Relations

Aspects of behavior

Communication

Encapsulation

Coordination

Attributes

Operations

Protocol

Trigger

Envelop

Task management
& synchronization

Speaker Notes

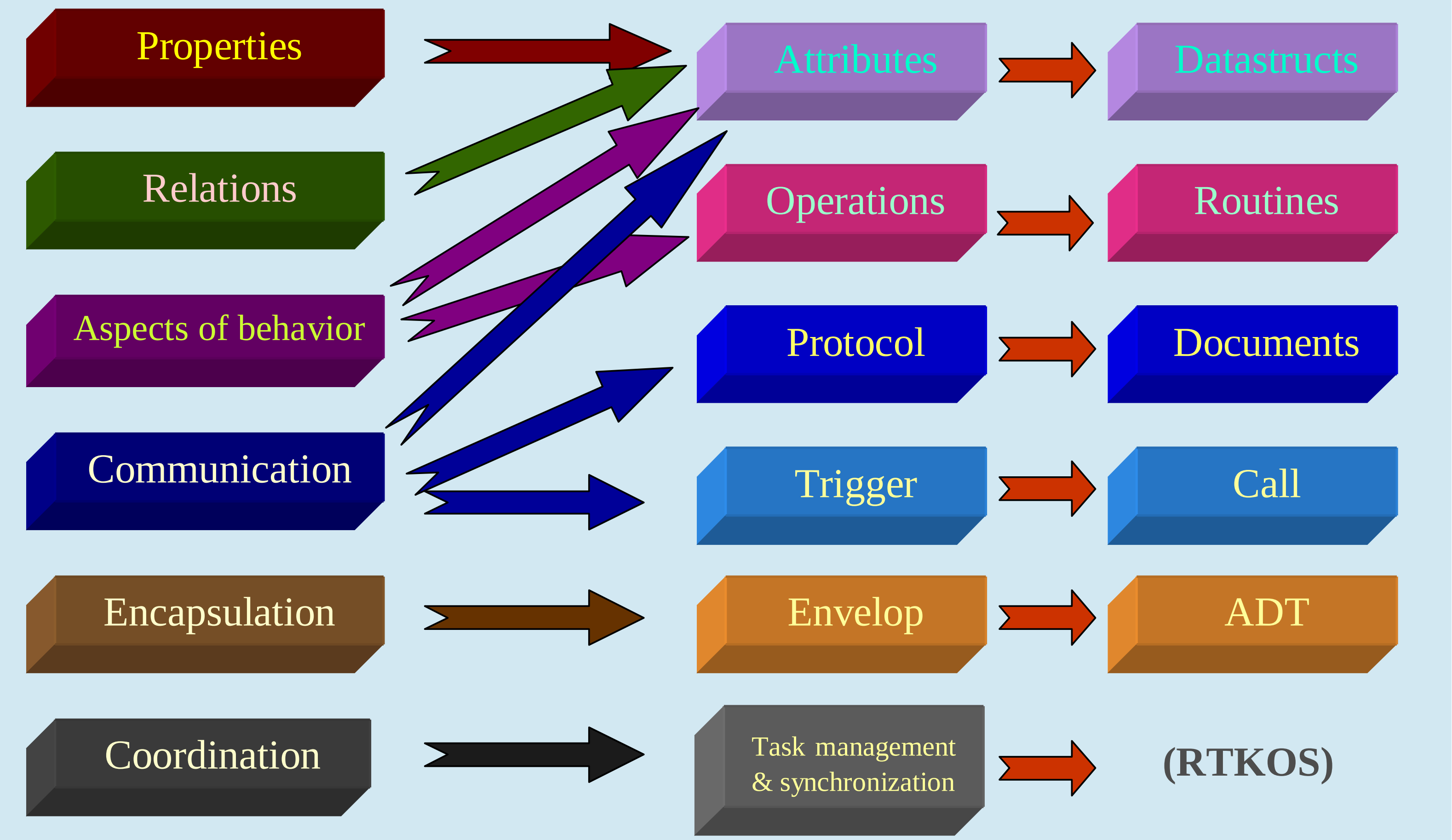
Where painters use colors and forms to generate an abstraction of their subject, programmers will use properties, aspects of behavior, relations, communication, encapsulation and coordination as ingredients for their model

The original, more natural modeling ingredients can be converted in a new set of mutually independent categories of modeling ingredients.

Operations are independent of the state of the individual. This state is represented by a set of attributes. Communication is a combination of three independent modeling elements: Attributes, protocols and the trigger that is caused at the receiver side. The communication path is a set of attributes, as is the transferred message or command.

The burden of keeping architectural views in concordance is minimal when the views are made as independent from each other as is reasonably possible. Describing the model in terms of independent modeling elements helps in keeping views independent.

Modeling => Implementation in SW



Speaker Notes

The original, more natural modeling ingredients can be converted in a new set of mutually independent categories of modeling ingredients. Programmers have straightforward implementations for each of these new ingredients.

Attributes are implemented in fields of datastructures, which in their turn are reserved areas in the available memory space.

Operations are independent of the state of the individual. This state is represented by a set of attributes. The routines implementing the operations use a reference to this set of attributes as an input parameter. In this way they themselves become independent of that state.

Speaker Notes

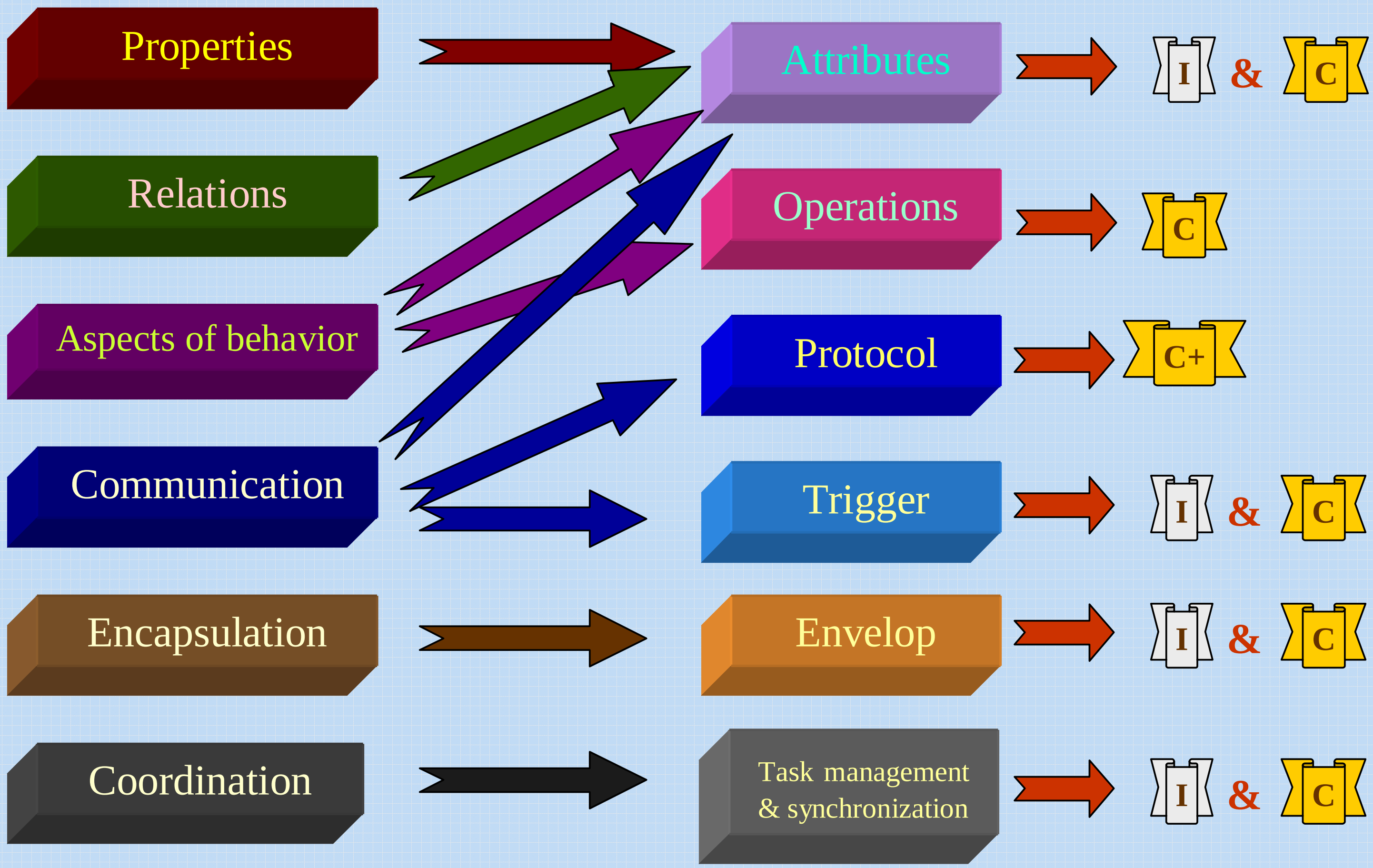
Protocols are defined by the language, which is used to program the operations and are further specified by the function prototypes of these routines.

A trigger represents the event of calling a operation. The attributes contained in the message are added as parameters to the call.

Encapsulation is achieved by applying abstract data types (ADT's).

Coordination is not supported directly by third generation languages. Instead it is implemented using services from a real time kernel operating system (RTKOS)

Wider Scope of Elements



Properties

Relations

Aspects of behavior

Communication

Encapsulation

Coordination

Attributes

Operations

Protocol

Trigger

Envelop

Task management
& synchronization

Speaker Notes

Where painters use colors and forms to generate an abstraction of their subject, programmers will use properties, aspects of behavior, relations, communication, encapsulation and coordination as ingredients for their model

The original, more natural modeling ingredients can be converted in a new set of mutually independent categories of modeling ingredients. Programmers have straightforward implementations for each of these new ingredients.

Some modeling elements have a wide scope. E.g. the scope of operations covers the class of items that share the operation as part of their behavior. In order to prevent Babylonic confusion the protocol used must have the widest possible scope. Ideally only a single scalable communication protocol should be used.

Metamodeling Elements

Meta modeling elements describe modeling elements

Type definition	**Class, attribute, parameter, operation**
Type	**Reference to type definition**
Interface type	**Describes a coherent set of operations**
Function prototype	**Special kind of type definition**

Metamodeling elements will have the highest chance to be reused

Speaker Notes

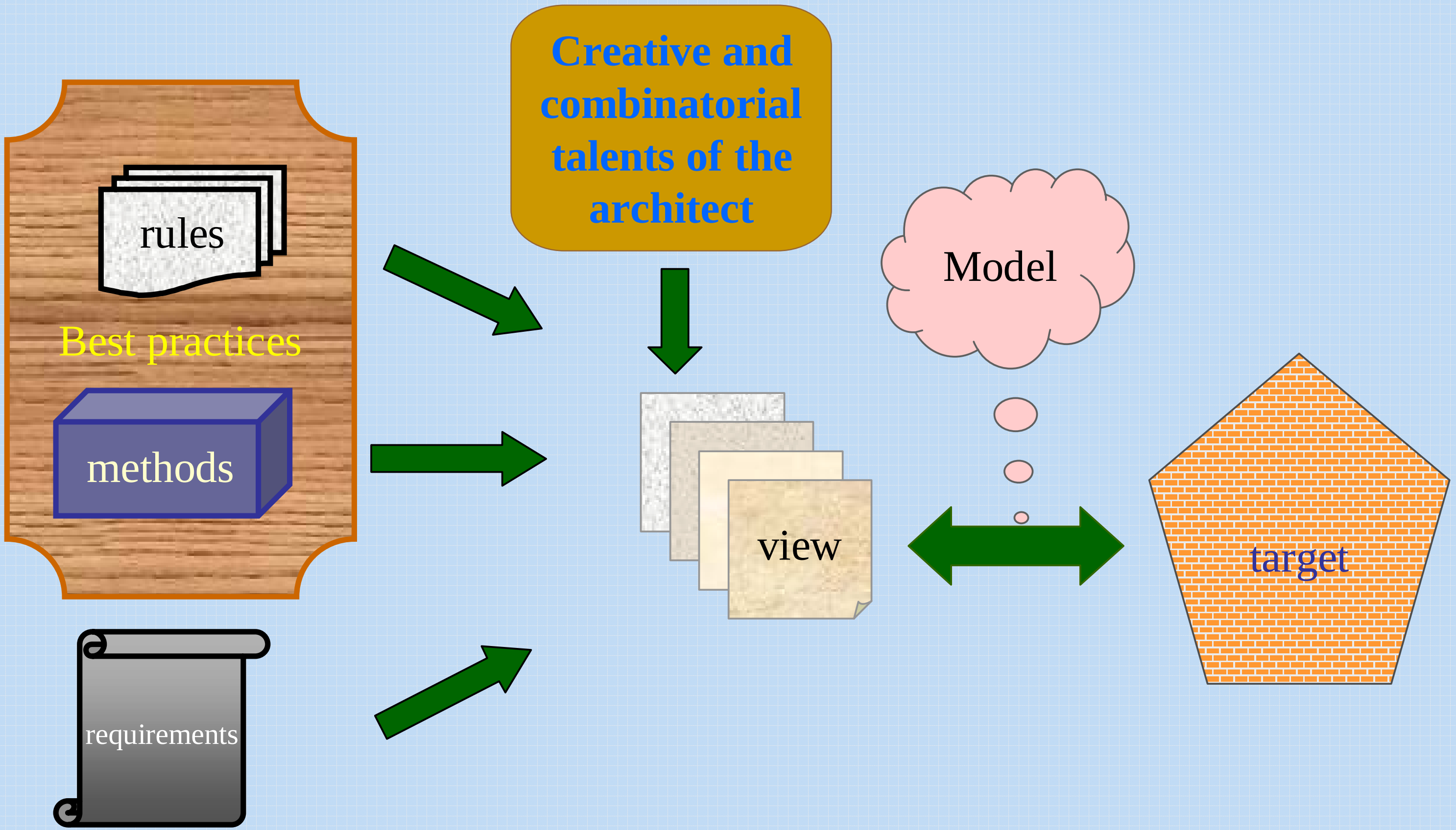
When a class of similar models is described precisely, then a definition of the corresponding type is given.

A type associates a model with its type definition.

An interface type is a description of part of a model. It can be specified and referenced independent of its encapsulating model.

A function prototype is a description of part of a method. It can be specified independent of its encapsulating model.

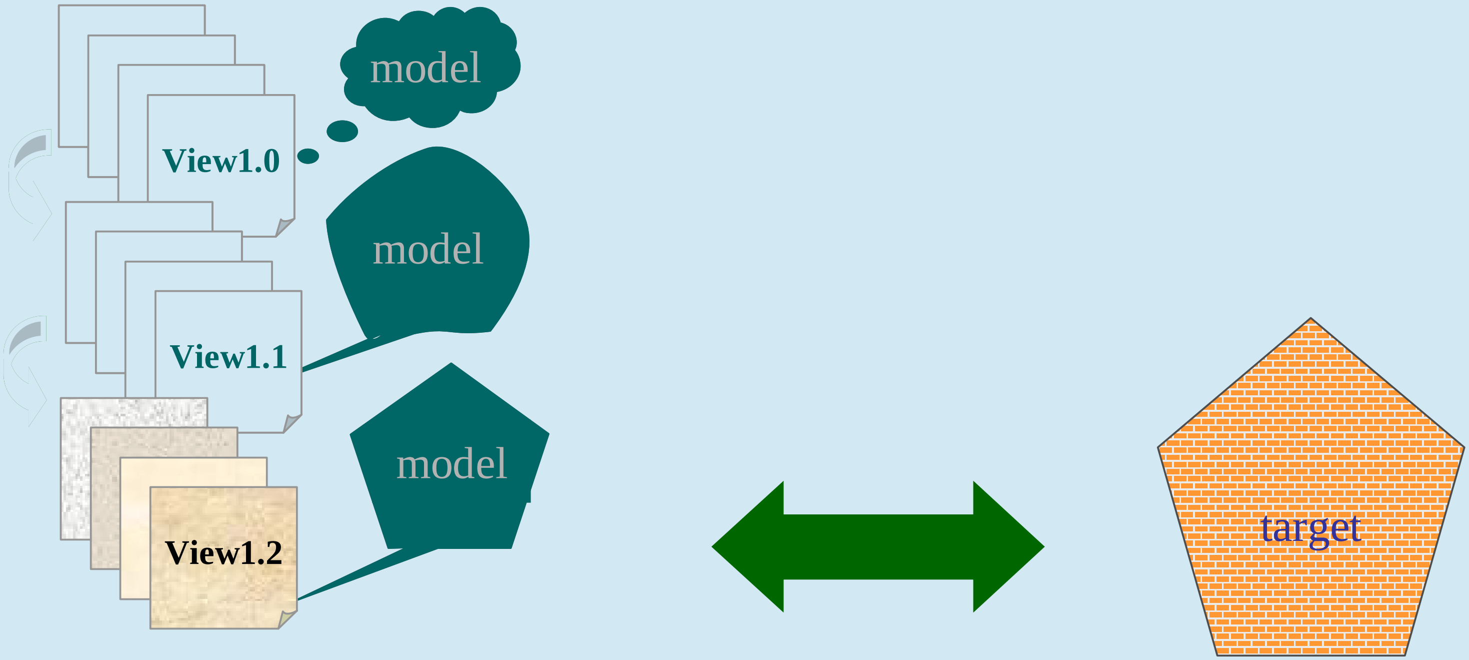
Architecture scope



Speaker Notes

An architecture is a set of views that are based on established rules and methods and on the customers requirements. The views represent different abstractions of the current model. The model itself is an abstraction or a partial realization of the required target product.

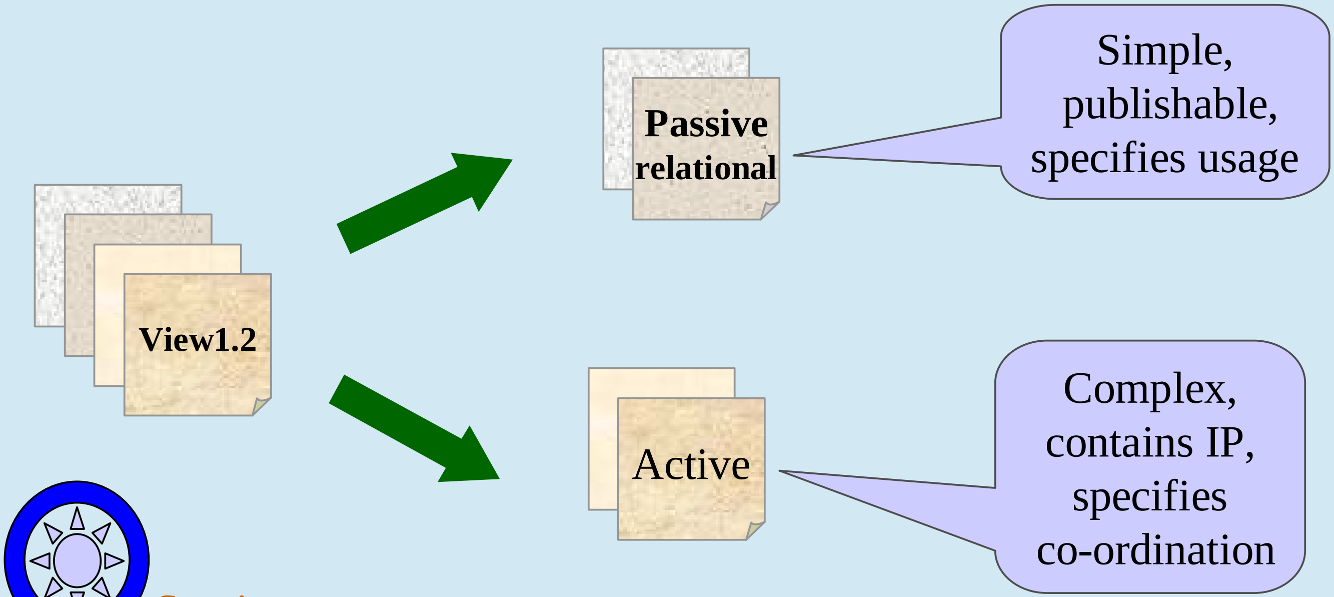
Architecture dynamics



Speaker Notes

An architecture is a set of views that changes dynamically with the progress of the underlying project. The views represent different abstractions of the current model.

Architecture Split up



Simple, publishable, specifies usage

Complex, contains IP, specifies co-ordination

Services

Speaker Notes

An architecture is a set of views that changes dynamically with the progress of the underlying project.

The set of views can be divided into two not completely orthogonal sub-sets:

A passive relational architecture part,

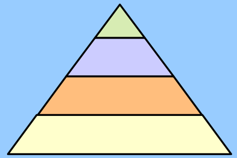
the dynamic architecture

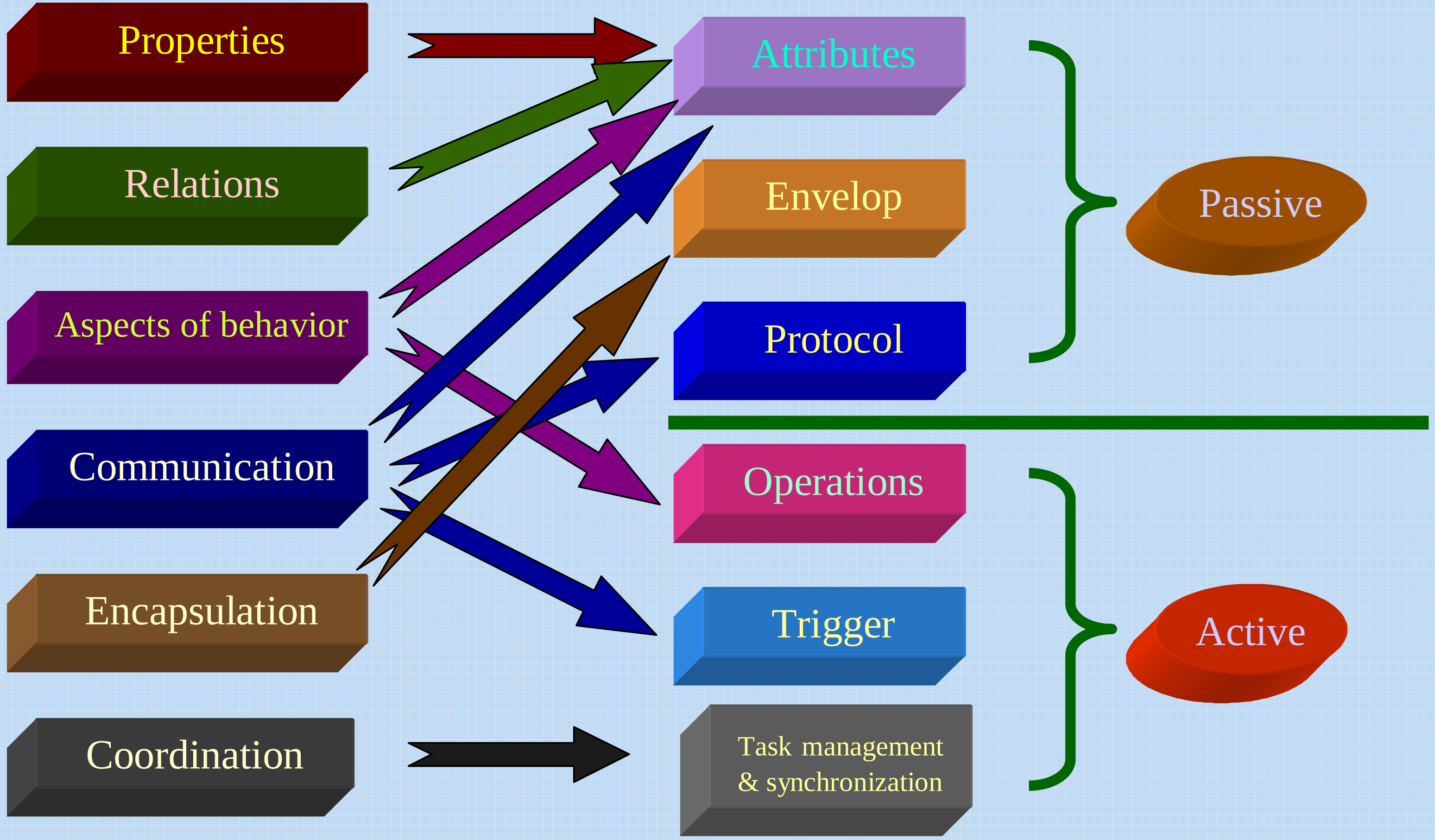
Apart from the local actors also some centralized or distributed services are part of the system. These services may be present in the surround or they may be created during system generation.

The passive relational architecture contains information that is publishable.

Specifying the passive relational architecture and specifying the necessary infrastructural support is orders of magnitude less complex than fully specifying the dynamic architecture.

Modeling => Division





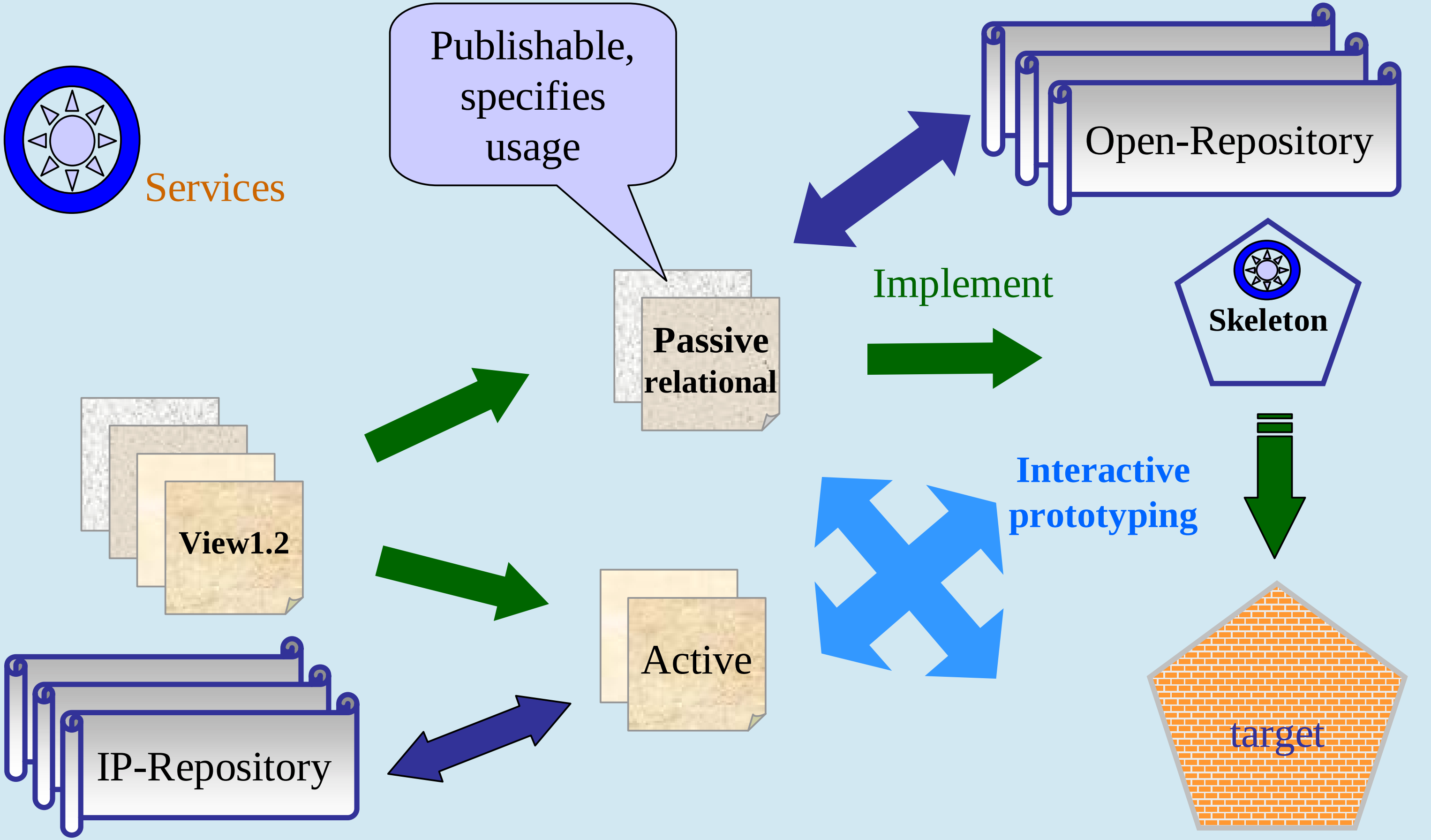
Speaker Notes

Where painters use colors and forms to generate an abstraction of their subject, programmers will use properties, aspects of behavior, relations, communication, encapsulation and coordination as ingredients for their model

The original, more natural modeling ingredients can be converted in a new set of mutually independent categories of modeling ingredients.

These independent modeling elements can be grouped into two categories. The first category represents the passive relational part of the architecture. The second category represents the active part of the architecture.

Exploiting the Division



9/30/2005

Speaker Notes

The passive relational architecture contains information that is publishable. It contains enough information to enable the generation of a skeleton prototype of the target in which the relational architecture can be fully tested. This skeleton prototype contains most of the supporting infrastructure. The generated or used infrastructure is fully functional. The other part is just a skeleton but it can be converted gradually and incrementally into a fully functional target product that fulfills all requirements.

Specifying the passive relational architecture and specifying the necessary infrastructural support is orders of magnitude less complex than fully specifying the dynamic architecture. Thus creating a skeleton on the basis of a passive relational architecture can give a project a head start where many of the risks can be estimated before the real and elaborate work starts.

Speaker Notes

It is possible to extend the skeleton in incremental steps until a fully functional target system is reached. In each of these steps a testable prototype supports detailed interaction between the requirements specification and the corresponding realization of these details in the target system.

Repositories exist in several categories. Here the open-repository is used to serve an open market. The IP-repository is used to exchange IP in a closed community.

Enabling Reuse

- The choice of communication protocols must be limited to **a single scalable protocol**
- Because:
  - **All individual components must be able to communicate with each other**
  - **Even the simplest component must implement all existing protocols**

Speaker Notes

If several different communication protocols are accepted, then extra resources must be spent to protocol conversion and to manage groups of components that can handle a chosen protocol.

If a single scalable protocol is supported then the component may negotiate its capabilities with its clients or servers, because it knows what scale of protocol it can handle. The corresponding resource needs are so small that it becomes acceptable for even the simplest component.

Scalable protocol means:

Direct link at the lowest scale

Dynamic link and/or script interpretation at medium scale

Distributed access at the highest scale

Reuse Promotion

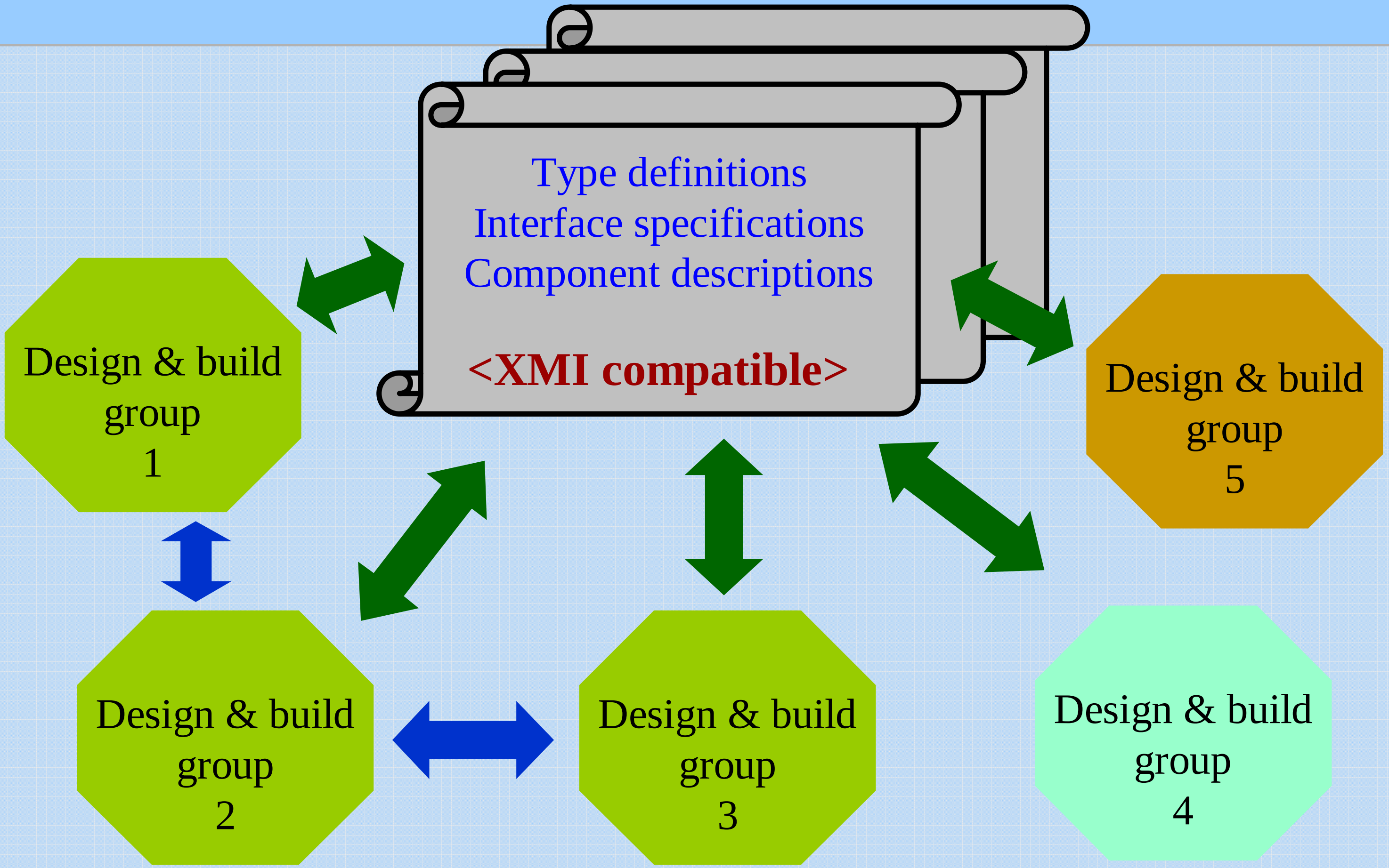
R

- Reuse must be promoted by **publishing design elements** (type definitions, interface definitions, component descriptions) **on repositories**.
- If this is done in machine retrievable way then an appropriate tool can construct **testable skeletons** of software modules from the retrieved data.
- The skeletons can be **integrated** with other components **in a testable prototype or simulation**.

Speaker Notes

Repositories can play a role equivalent to the current role of module handbooks. Besides of that they pose the possibility to offer their information in a machine retrievable way. This again opens the possibility to automatically create skeletons from the retrieved data. Such skeletons can then be applied in running and testable prototypes or simulations of the target product.

Reuse Promoting Repositories



Type definitions
Interface specifications
Component descriptions

<XMI compatible>

Design & build group 1

Design & build group 2

Design & build group 3

Design & build group 4

Design & build group 5

Speaker Notes

A series generally accessible sites (repositories) must support reuse of existing types, interfaces and components

Design & build groups may inquire the sites for type definitions and interface definitions

The site might also specify the (passive relational) architectural design and provide descriptions of complete packages of software components and may provide information about the the suppliers of these components.

Tools may use the information retrieved from a repository for the creation of skeletons of modeling blocks. These skeleton blocks may range from skeletons of interfaces to skeletons of complete components. The information suffices to build a working and testable prototype of the target application. This prototype can be converted in incremental steps into a fully functional system.

Repositories

- Support archival and retrieval of XML based scripts that contain design elements.
- Represent the equivalent of module handbooks.
- The XMI standard guards exchangeability of data between disparate tools.
- Repositories are essential for creating and supporting an open market.

Speaker Notes

Repositories can play a role equivalent to the current role of module handbooks. Besides of that they pose the possibility to offer their information in a machine retrievable way. This again opens the possibility to automatically create skeletons from the retrieved data. Such skeletons can then be applied in running and testable prototypes of the target product.

The XMI standard proposed by the OMG secures easy and reliable exchange of the design elements that are archived on the repository.

Together this may cause a busy open market for software components. The same approach can be used both for hardware and software components. Brought together it will enable a market for hybrid components.

eXtensible Markup Language

- Derivative of SGML
- More flexible than HTML
- Not restricted to Internet
- Plain text based protocol
- Prepared by ISO W3C
  - http://www.w3.org
- Accepted by ISO in February 1998
- Empowered by a series of associated standards
- Most of them are still in preparation

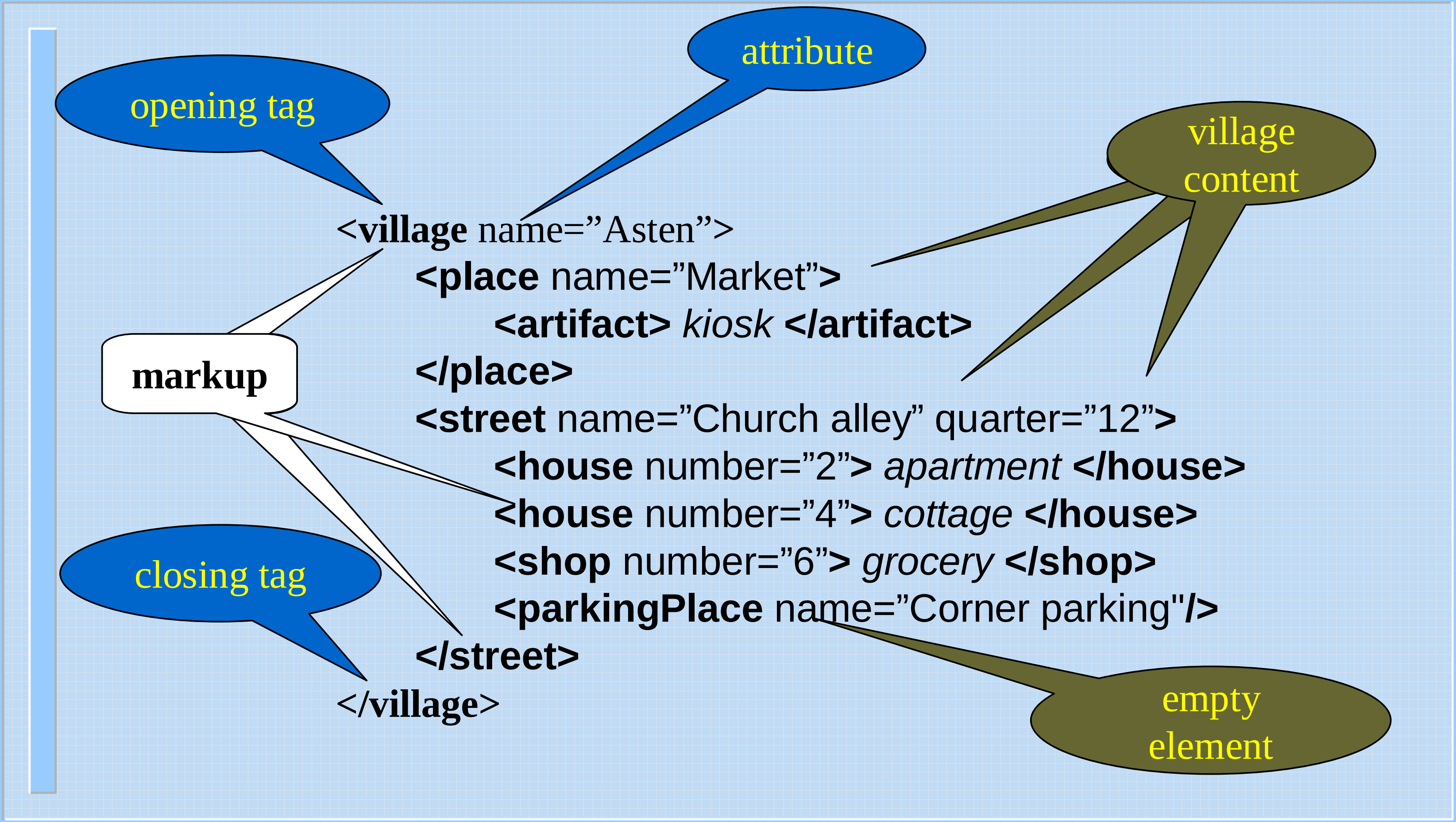
Speaker Notes

XML is a derivative of SGML. SGML is used for example to design and print handbooks. HTML is also a derivative of SGML but is is too limited for the current for flexibility in data exchange via web pages.

Its use is not restricted to the Internet. Many tools already use XML files for the exchange of structured information with other tools.

XML is empowered by a series of associated standard file types and language definitions. Most of these standards are still in preparation. Currently the structure of XML files is defined using DTD files. DTD files are an inheritance of SGML. In the future schema files will be used. Schema files offer a much more detailed definition of the structure of an XML file. Apart from that schema files themselves are XML files.

XML As a Container



```
<village name=”Asten”>
      <place name=”Market”>
            <artifact> kiosk </artifact>
      </place>
      <street name=”Church alley” quarter=”12”>
            <house number=”2”> apartment </house>
            <house number=”4”> cottage </house>
            <shop number=”6”> grocery </shop>
            <parkingPlace name=”Corner parking"/>
      </street>
</village>
```

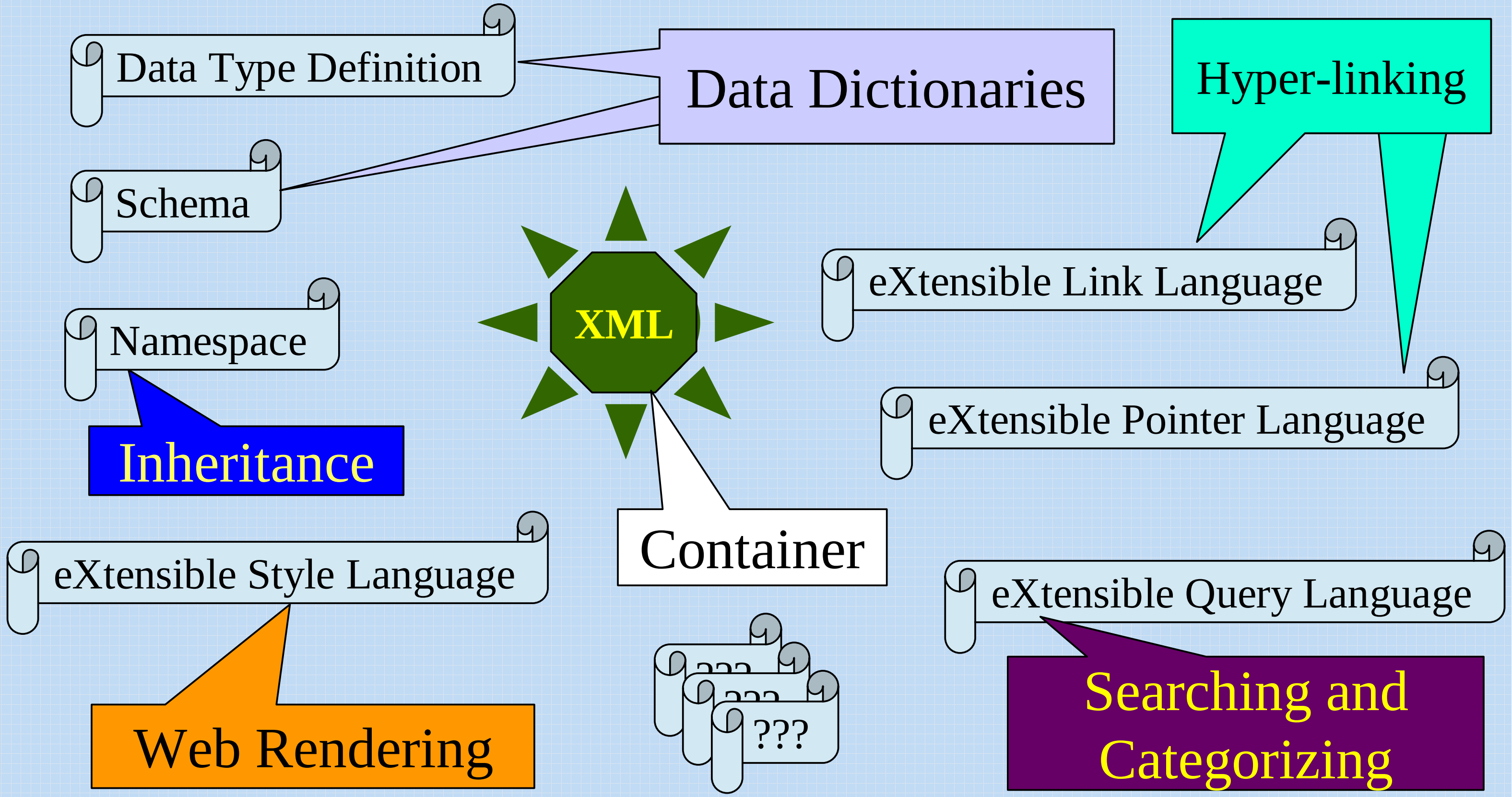
Speaker Notes

The markup in XML files consists of tags. These tags can be freely chosen. However the XML file must be well formed. With non-empty elements each opening tag must be followed by a closing tag. The tags may incorporate one or more attributes. The tags and the structure of their content may, but must not be defined in a corresponding DTD or schema file.

Tags belong to a namespace. This may be the default namespace as in the example or it may be a special namespace:

```
<author name="Hans van Leunen">
  <title>Ir</title>
  <paper:title>Architecture of component-based systems</paper:title>
</author>
```

Associated File Types



Data Type Definition

Schema

Data Dictionaries

Hyper-linking

eXtensible Link Language

eXtensible Pointer Language

Namespace

Inheritance

XML

Container

eXtensible Style Language

Web Rendering

???

???

???

eXtensible Query Language

Searching and Categorizing

Speaker Notes

The standards that are associated with the XML standard make XML files into containers that have much in common with databases.

DTD and schema files play the role of data dictionaries. XQL files enable searching through XML documents and categorizing of sections of XML documents. XSL files enable the rendering of the data contained in XML files on web pages. XLL and XPL help interlinking XML documents and sections of XML documents. The namespace standard enables inheritance between data dictionary files.

The list of associated standards will not stop here. Many other associated standards are already proposed.

Comparison

- Databases
  - Optimized for archival and retrieval of **series of similar types**
- XML
  - Optimized for archival, retrieval and exchange of **hierarchically structured data**

Speaker Notes

XML files have much in common with databases. Apart from that there are also important differences. The main difference is that databases are much better suited for handling data that occur in large series of similar types, while XML is much better suited to handle hierarchically structured data.

Another difference is that XML is well suited for streaming data between applications.

XML Metadata Interchange

XMI

- **XMI** is a standard that is prepared by the Object Management Group (**OMG**)
  - http://www.omg.org
- **XMI** covers:
  - Meta-meta-modeling ⇒ XMI script
  - Meta-modeling ⇒ MOF script
  - Modeling ⇒ UML script

Speaker Notes

The Object Management Group has published its second version of the proposal for the XMI standard.

The XMI standard describes how design information must be stored, retrieved and interchanged.

For that reason the OMG publishes DTD files for the XML scripts that must be used to describe design elements in the Meta Object Facility (MOF) and in the Unified Modeling Language (UML).

It also specifies how the corresponding XML files must be generated.

XMI

XMI

- **XMI** uses **XML** as scripting language
- The Meta Object Facility (**MOF)** is used for specifying meta-models, such as **IDL** files
- Unified Modeling Language (**UML**) is used for designing systems
- **XMI** specifies **DTD** files for defining meta-model and model data exchange scripts

Speaker Notes

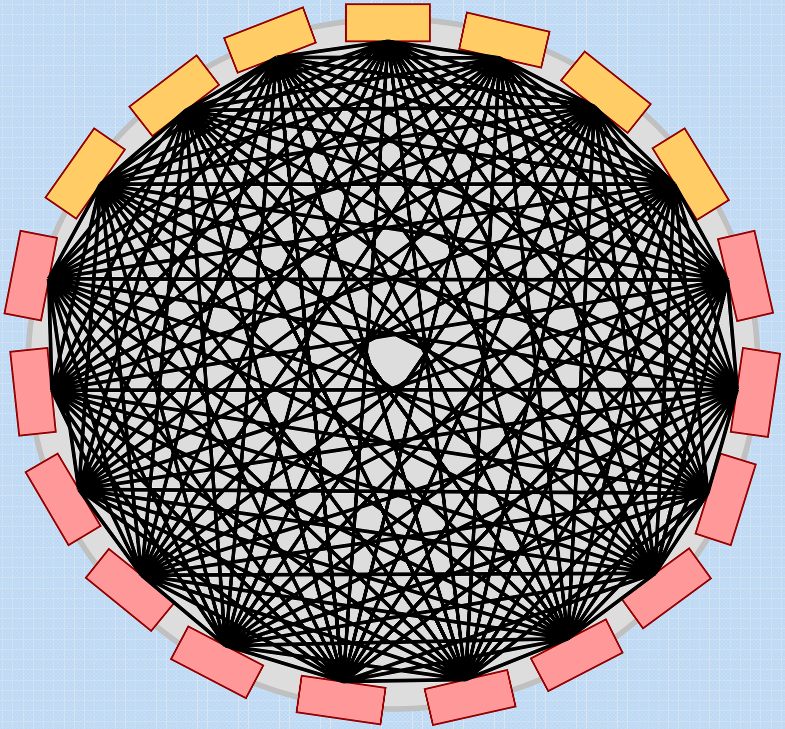
The Object Management Group has published its second version of the proposal for the XMI standard.

The XMI standard describes how design information must be stored, retrieved and interchanged.

For that reason the OMG publishes DTD files for the XML scripts that must be used to describe design elements in the Meta Object Facility (MOF) and in the Unified Modeling Language (UML).

It also specifies how the corresponding XML files must be generated.

Relational Complexity in Monolithic System or Part



n(n-1)/2 potential relations

n = Nr of related items

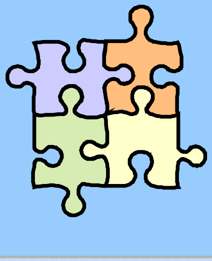
Speaker Notes

Let a system be implemented by n operations, which together work on m modelling elements. r of the controlled modelling elements are relations. A novice that has to understand the implementation is confronted with ½ × n × (n-1) potential relations between the operations and with n × m potential relations between the operations and the controlled modelling elements. Relations may differ in character. Say that there exist t different types of relations. Assuming that the modelling elements that represent relations do not describe relations between relations, the potential relational complexity, to which the novice is confronted, equals:

$(\frac{1}{2} \times n \times (n-1) + n \times m + r) \times t$

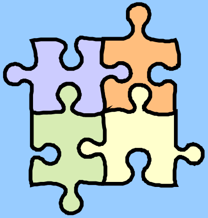
The same complexity is encountered by reverse engineering tools or when a bug has to be resolved that cannot be directly related to a single design element. With other words: potential relational complexity is a good characteristic for the design related difficulties that can be encountered in managing the current project.

In numbers



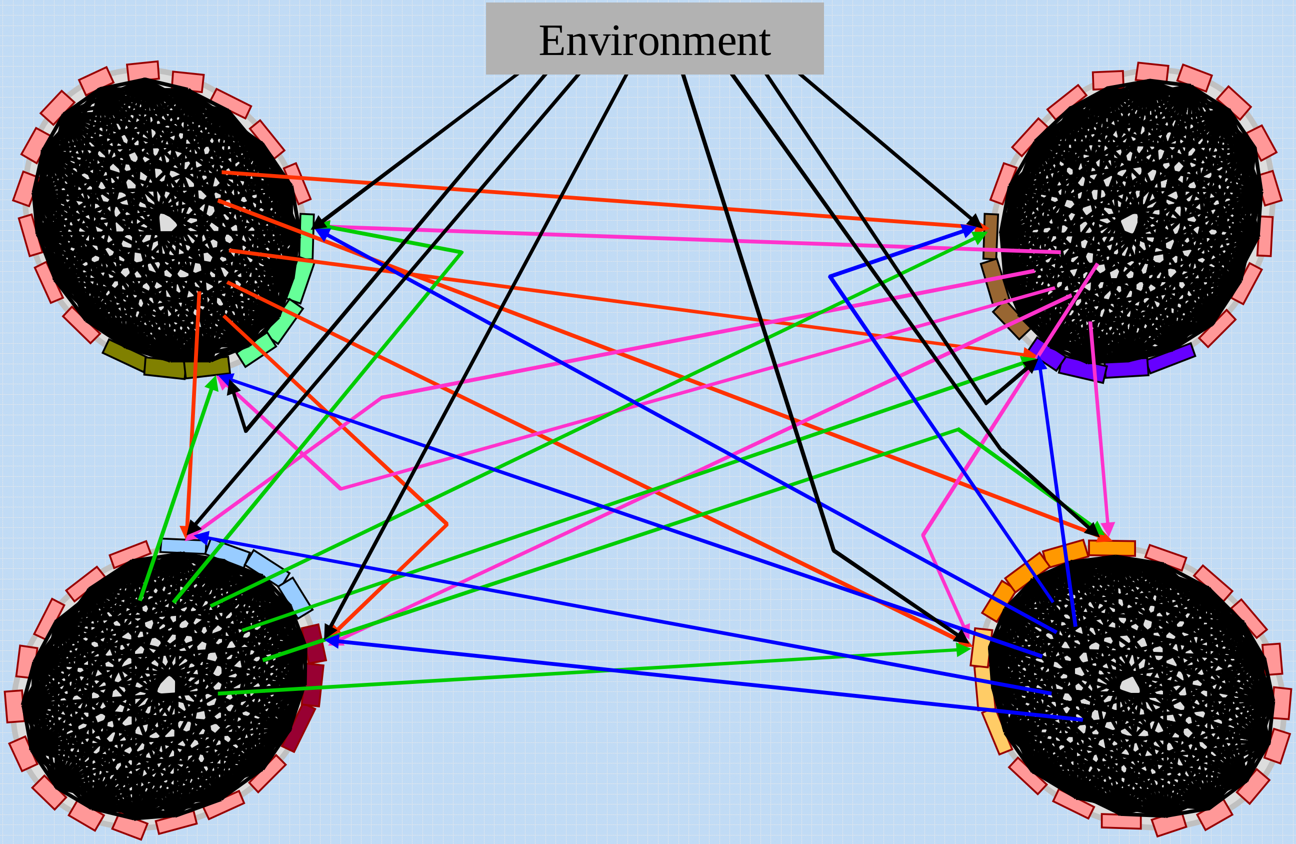
- 100 items ⇒ 99 • 100 potential relations
- 1000 items ⇒ 999 • 1000 potential relations
- 10 modules containing 100 items ⇒ maximally 99 • 100 potential relations inside a module plus 9 • 10 relations between modules.
- Nobody sees more than 9990 relations!!!

Impact



- The reduction of the potential relational complexity has the largest impact when systems are completely built as component based systems
- Currently all systems that apply software components apply components in a relatively small subsystem
- This is why nobody has got a proper feel for the real power of software component technology

Complexity in Component Based System



Easily two orders of magnitude better than monolithic case

Speaker Notes

A component typically contains one up to five interfaces and each interface contains typically between two and ten operations. Thus a component contains typically between ten and fifty operations and it has between two and twenty attributes. Typically about twenty different types are used. So internally the potential relational complexity of a component with
(n = 25; m = 10; r = 4; t = 20) ⇒ (25/2*(25-1)+25*10+4)*20 = 11080
is of the order of ~104. The potential relational complexity for the system integrator that configures the system out of 10 reusable components with
(n = 3 × 10; m = 0; r = 40; t = 25) ⇒ (30/2*(30-1)+30*0+40)*25 = 11875
is of the order of ~104. An equivalent monolithic design would have a potential relational complexity of ~106. A system consisting of one hundred reused components would have a relational complexity given by:
~(n = 3 × 100; m = 0; r = 400; t = 25) ⇒ ~(300/2*(300-1)+300*0+400)*25 = 1131250.

Speaker Notes

This is in the order of ~106. A corresponding design without components would have a relational complexity of ~108. Together the reused components have a potential relational complexity of the order of ~106. However, since the creation of the components will be delegated to different design groups, a single developer never encounters that complexity. This proves that the component oriented approach has a very significant beneficial effect on the potential relational complexity.

Layering also has a beneficial influence on relational complexity. For example a four layered system has a potential relational complexity that is 30% better than a monolithic system. This is still far from what can be reached with component technology.

Depending on the efficiency at which encapsulation is pursued and on the depth of the inheritance the potential relational complexity of an object oriented class library can range from worse than an equivalent monolithic system to as good as an component based design.

Potential relational complexity, or better its antonym relational clarity, has a direct correspondence with the manageability of the design.

If Reuse Must Be Optimized

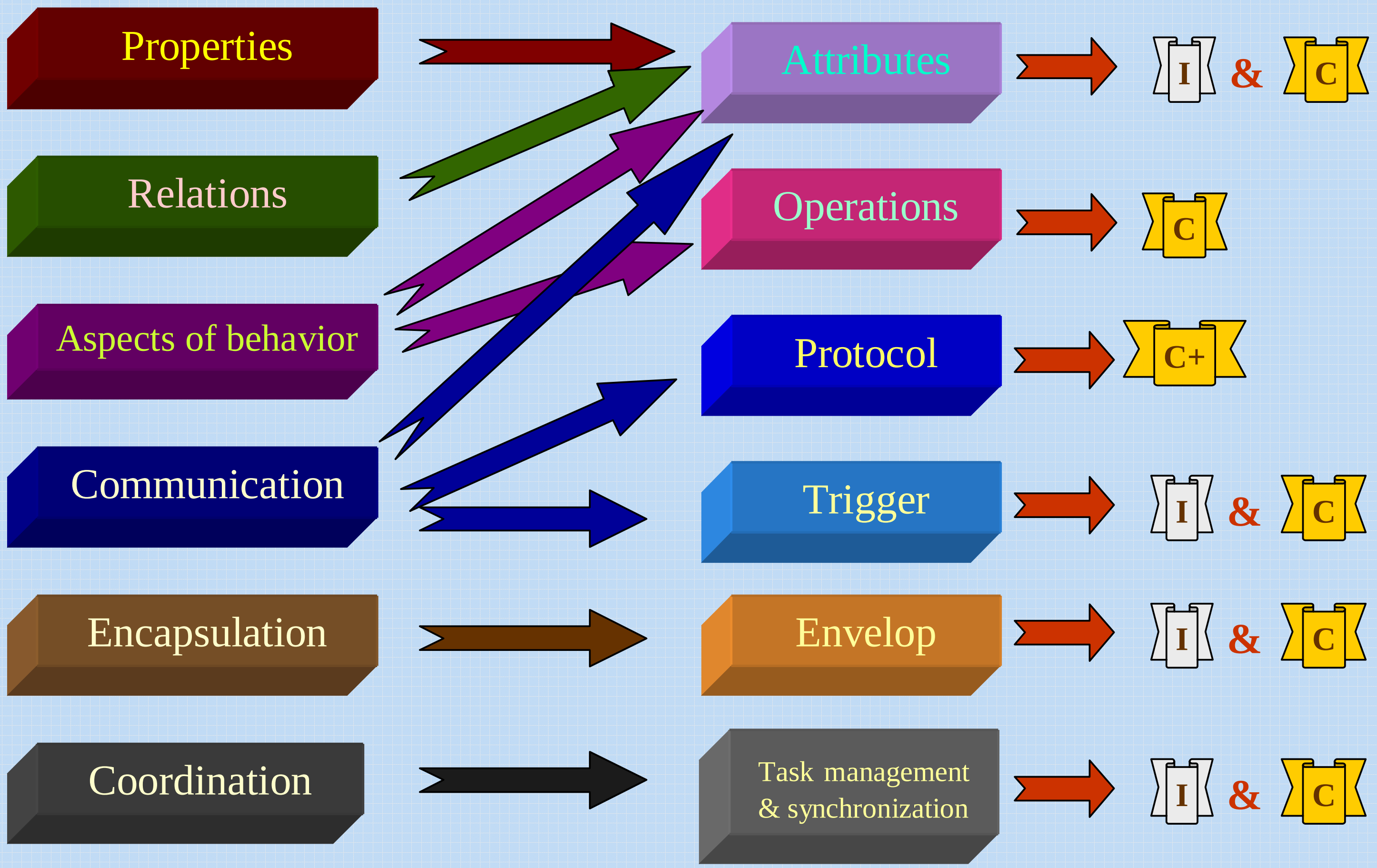
- **Analyze the complete application field**
- **Bring order in the models found there**
- **Find classes of equivalent functionality**
- **Exploit class wide aspects**
- **Exploit the ranking of complexity**
- **Group methods into intuitive interfaces**
- **Encapsulate instances**
- **Hide internals**

Speaker Notes

A procedure is presented according to which reuse of efforts in the software design and build process can be optimized. Many of the steps listed here on itself already provide some sub-optimization.

Driving steps too far may hamper the installment of other steps. For example inheritance exploits the ranking of complexity. Driving it too far may hamper proper encapsulation.

Wider Scope of Elements



Properties

Relations

Aspects of behavior

Communication

Encapsulation

Coordination

Attributes

Operations

Protocol

Trigger

Envelop

Task management
& synchronization

I & C

C

C+

I & C

I & C

I & C

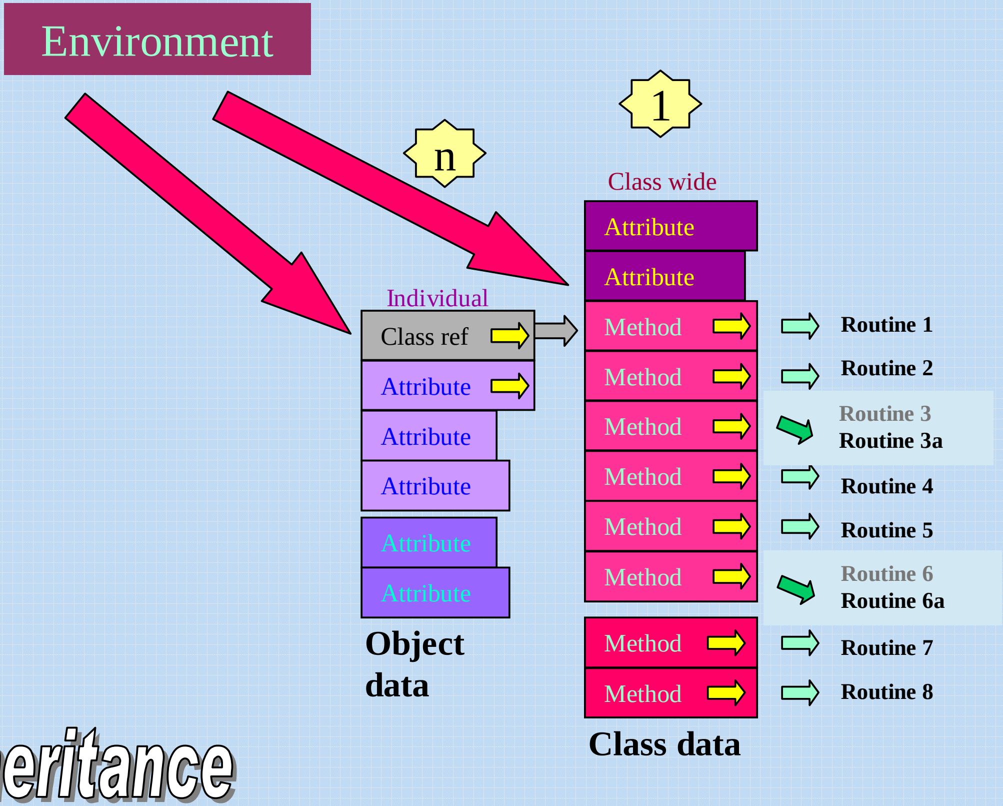
Speaker Notes

Where painters use colors and forms to generate an abstraction of their subject, programmers will use properties, aspects of behavior, relations, communication, encapsulation and coordination as ingredients for their model

The original, more natural modeling ingredients can be converted in a new set of mutually independent categories of modeling ingredients. Programmers have straightforward implementations for each of these new ingredients.

Some modeling elements have a wide scope. E.g. the scope of operations covers the class of items that share the operation as part of their behavior. In order to prevent Babylonic confusion the protocol used must have the widest possible scope. Ideally only a single scalable communication protocol should be used.

ADT→object Orientation



Environment

n

1

Individual

Class ref

Attribute

Attribute

Attribute

Attribute

Attribute

Object data

Class wide

Attribute

Attribute

Method

Method

Method

Method

Method

Method

Method

Method

Class data

Routine 1

Routine 2

Routine 3

Routine 3a

Routine 4

Routine 5

Routine 6

Routine 6a

Routine 7

Routine 8

ADT++

Inheritance

Redirected

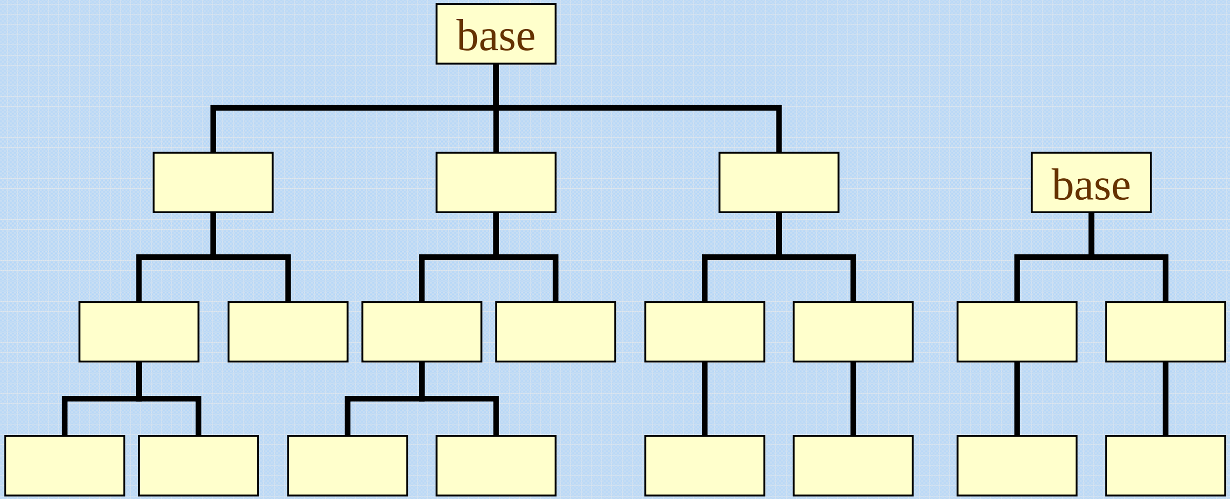
Speaker Notes

The individual and the class to which it belongs are both represented by a set of attributes that are contained in a datastructure. The datastructure of the individual contains a reference to its class. The class datastructure contains a list of references to routines that implement the methods. Together they form a cluster that represents the complete functionality of the individual.

Inheritance is implemented by extending the tables of attributes or routine references contained in the datastructures and by redirecting the references to the routines that implement the behavior to routines that are more sophisticated or that use the extra attributes contained in the extended datastructures.

This shows how child classes can be derived from parent classes. Multiple inheritance is implemented by exploiting the possibility to aggregate instances of other classes into an enveloping class and then exporting the access to the functionality of the contained object. (Multiple inheritance is not shown here.)

Class Libraries



Class libraries are structured sets of class modules

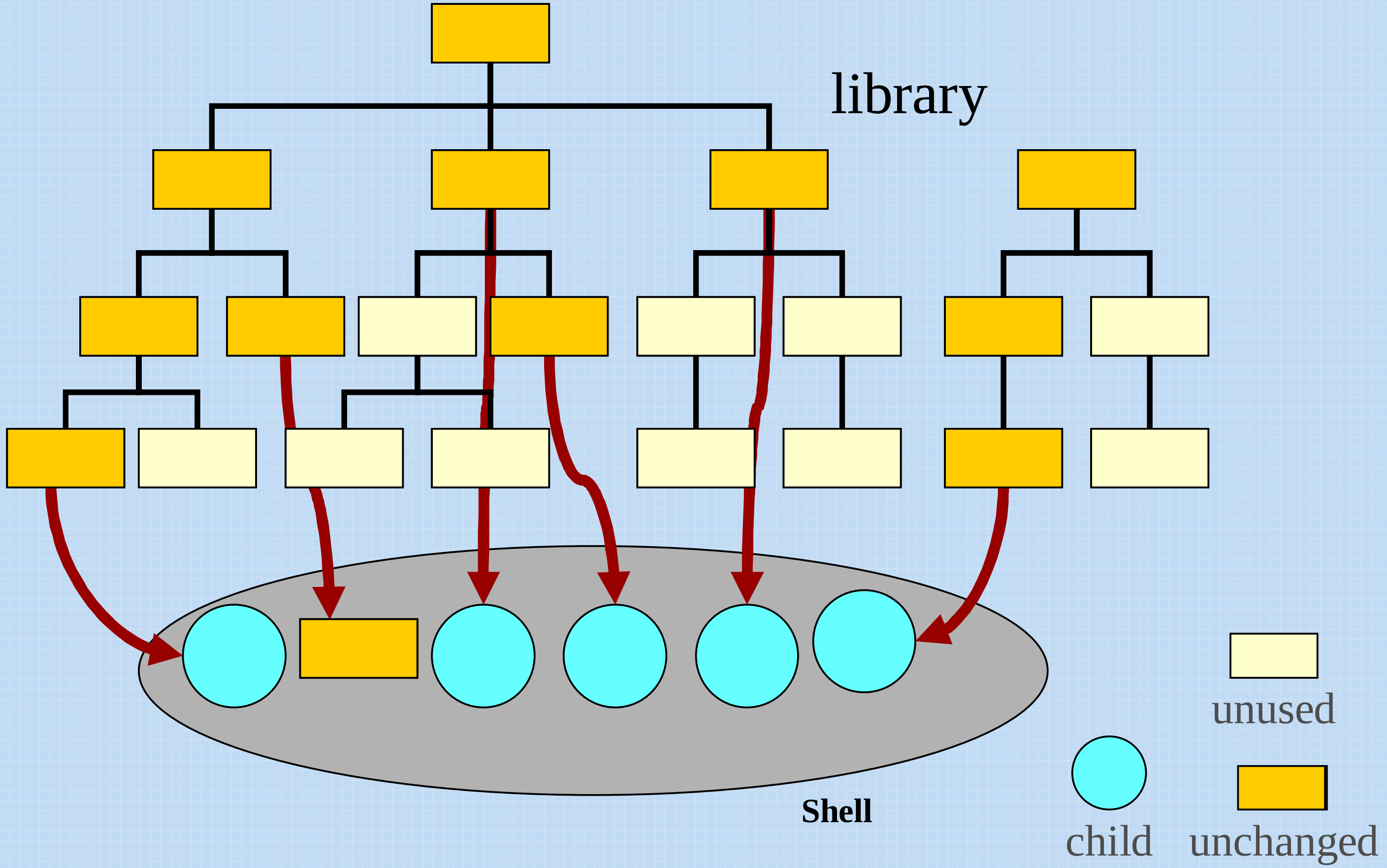
Class modules contain differences with respect to parent class

Base class modules contain the full class

Speaker Notes

Class libraries are ordered sets of class modules. Class modules contain the differences with respect to their parent class. The modules of the top classes in the hierarchy contain the complete class. If inheritance is confined to single inheritance, the library will have a hierarchical structure. Otherwise a network structure results.

Applications From Class Libraries



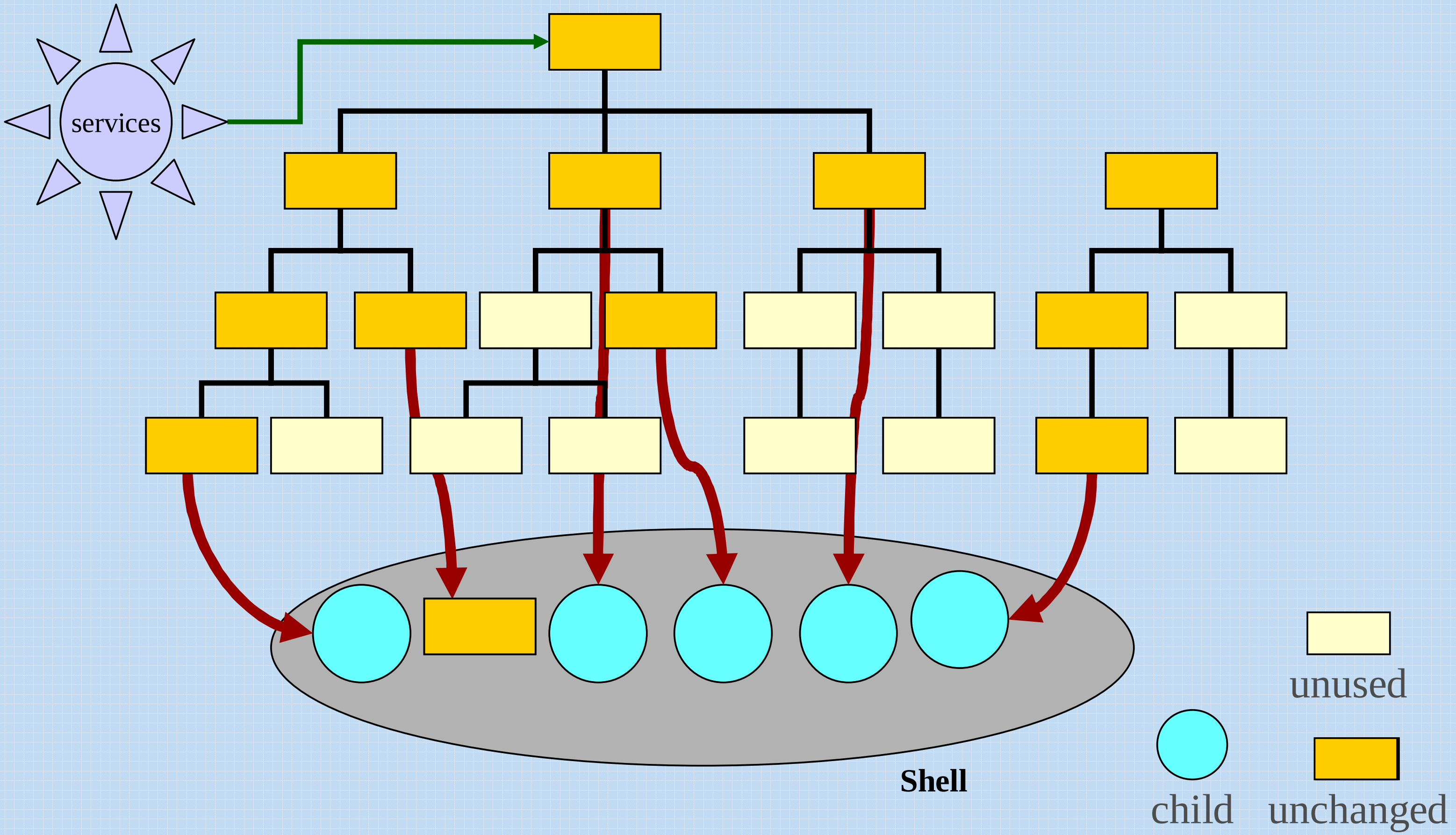
Speaker Notes

When generating applications from a class library then new classes are derived from classes present in the library, even if only a slight change to that class would be required. This is done in order to prevent that the existing class library gets disturbed. Only when a class can be used unchanged it may be added directly to the application.

Usually not all of the class library is used. Often there are no efficient cleanup tools available that remove all unnecessary code. Usually the linker removes unused classes but cannot remove unused class members. Due to the inheritance relations it is difficult to componentize the class library such that the amount of unnecessary code is minimized.

In order to be able to work efficiently with a class library, a programmer must have the the source code of the library at his disposal. Without such access he cannot optimize reuse. There is a real chance that he must completely re-implement most of the extended methods. Also debugging may become problematic without access to source code. Access to source code exposes all intellectual property that is invested in a class library to the users of that library.

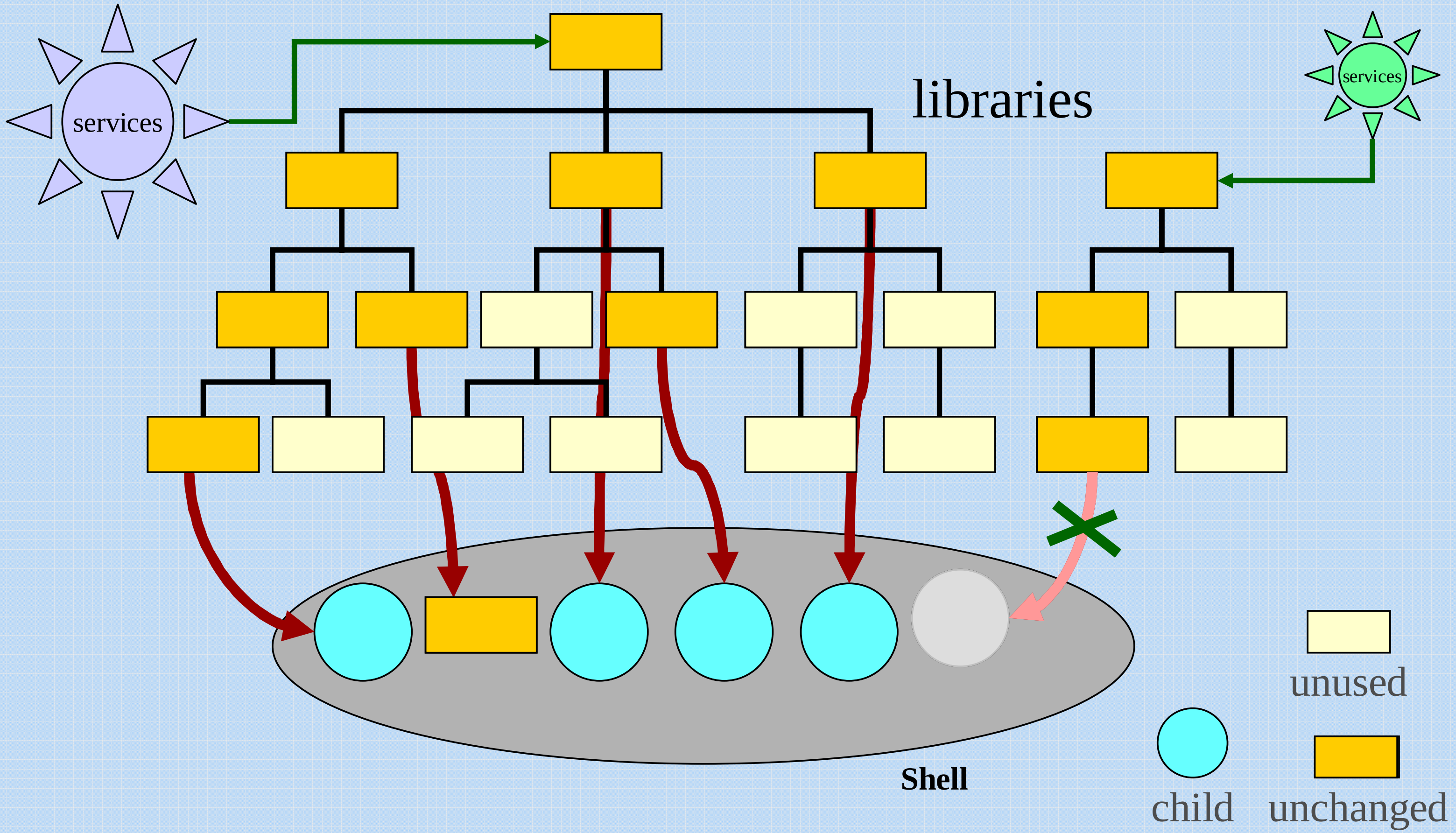
Implementing Distributed Services



Speaker Notes

It feels natural to offer distributed infrastructural services by installing them in top level classes. In that way they ripple through all lower level classes and into all applications that are built with that library. If use is made of these services, then in all of these occasions interdependencies are raised. This may render it difficult to change the implementation of these services at a later date. In the extreme case it may render the complete library archaic.

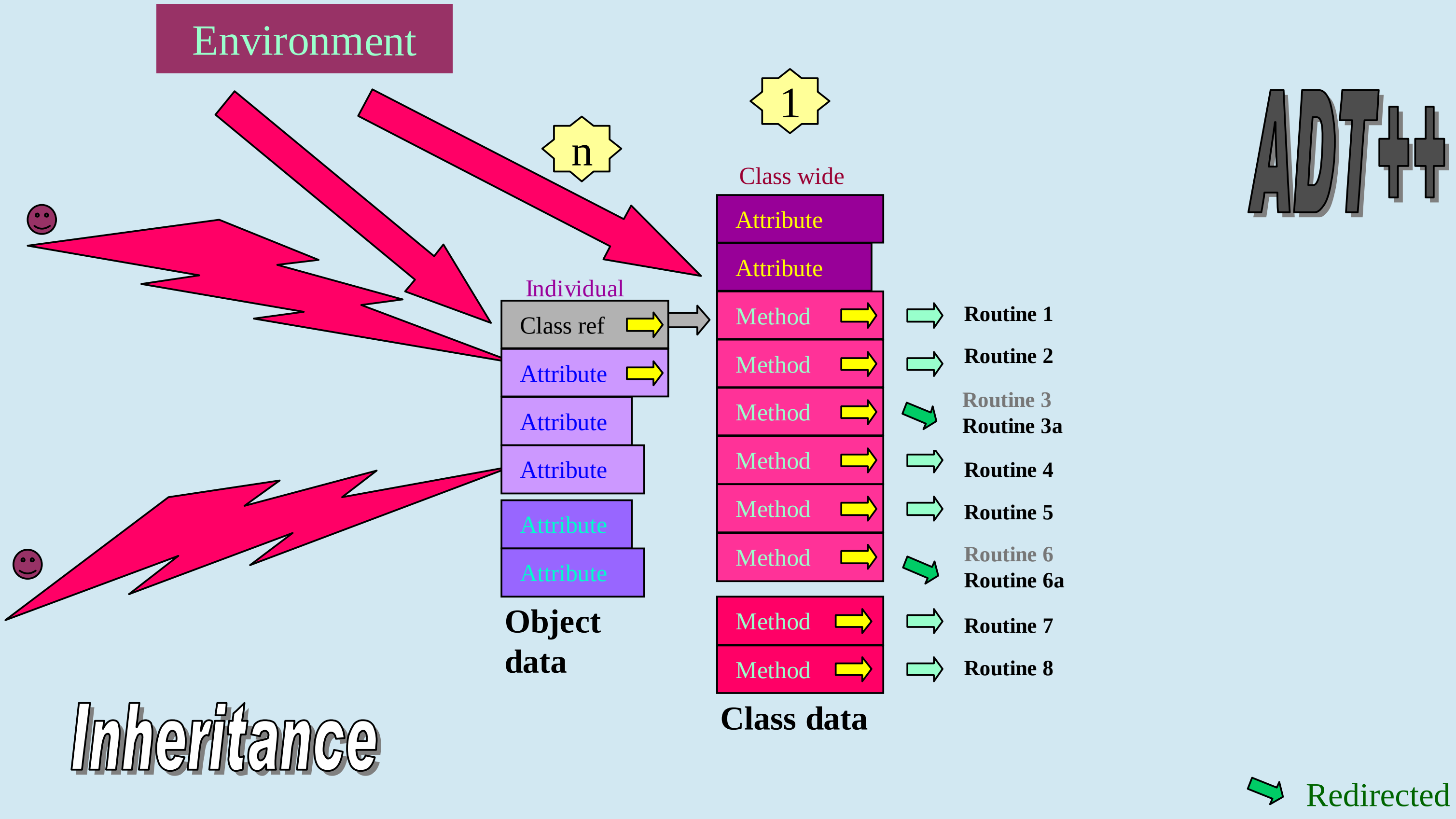
Distributed Services Conflict



Speaker Notes

If in two or more different class libraries the infrastructural services are implemented in different ways, while in both cases the services are implemented in top level classes, then sub-libraries of one of these class libraries cannot be combined with other class libraries of that set.

Object Orientation, Deficiencies



Environment

n

1

ADT++

Class wide

Attribute

Attribute

Individual

Class ref

Attribute

Attribute

Attribute

Attribute

Attribute

Method → Routine 1

Method → Routine 2

Method → Routine 3 Routine 3a

Method → Routine 4

Method → Routine 5

Method → Routine 6 Routine 6a

Method → Routine 7

Method → Routine 8

Object data

Class data

Inheritance

Redirected

Speaker Notes

Most object oriented languages are careless with encapsulation. Often direct access to attributes is tolerated. This inhibits establishment of uniform access and makes it impossible to make the class instance fully responsible for its behavior.

Object Orientation, Deficiencies

- Most object oriented systems are sloppy with encapsulation
- Multiple communication protocols possible
- Deep inheritance may cause unwanted dependencies
- Distributed services appear everywhere with their positive and with their negative effects

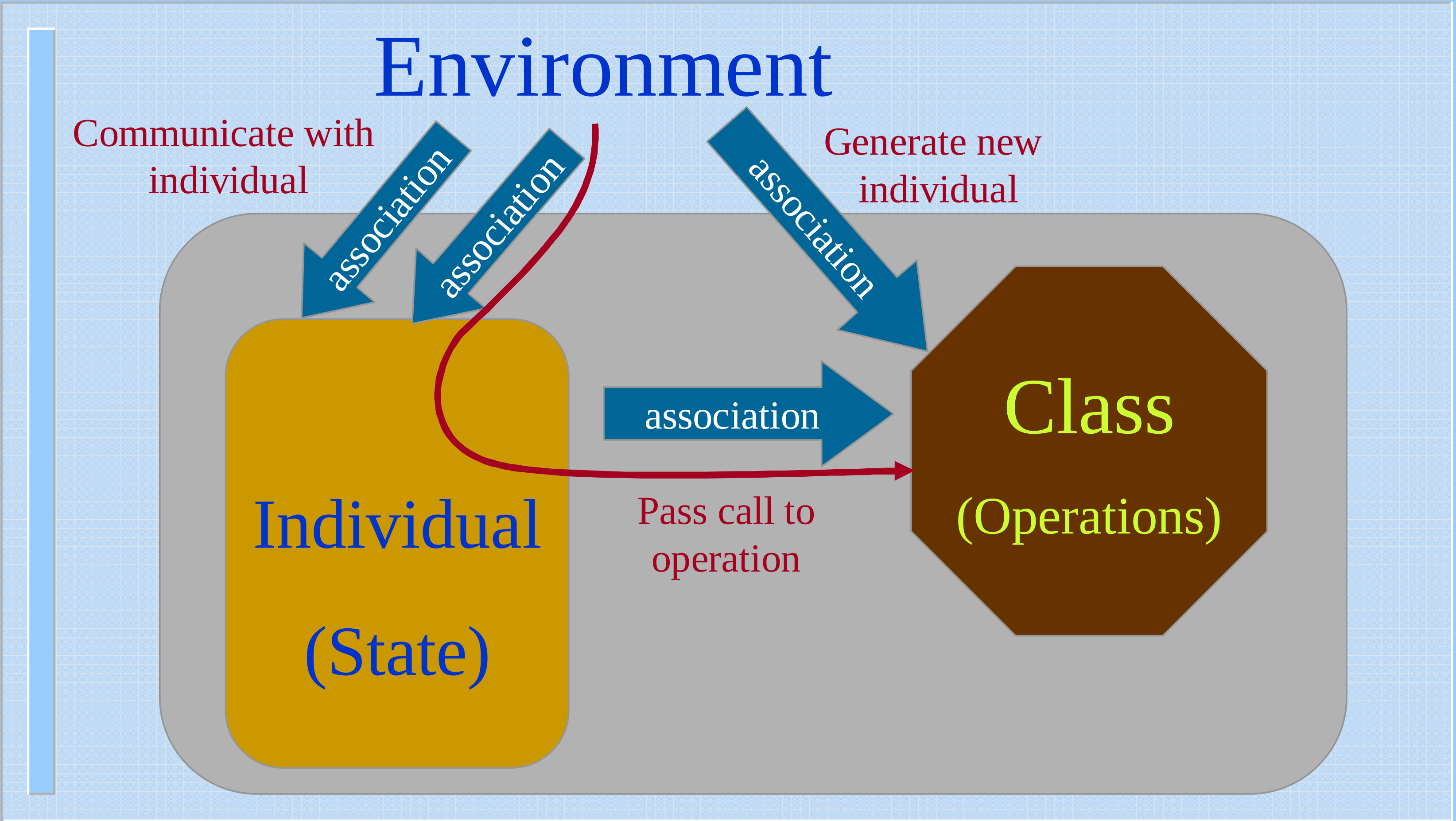
Speaker Notes

Most object oriented languages are careless with encapsulation. Often direct access to attributes is tolerated. This inhibits establishment of uniform access and makes it impossible to make the class instance fully responsible for its behavior.

If multiple ways of accessing data are tolerated then this effectively means that multiple communication protocols are supported. This hampers proper access management.

Any service that is implemented in a top level class will ripple through all lower level classes and will appear in all applications that are built with that class library. This may cause unwanted dependencies. It may render the library archaic and it may render class libraries mutually incompatible.

Encapsulation



Environment

Communicate with individual

Generate new individual

Pass call to operation

Speaker Notes

Operations are modeling elements that have a class-wide scope. Communication has an even wider scope. Associations are used as a communication path in order to pass messages and commands.

Attributes can be private to the individual or they have a class-wide scope

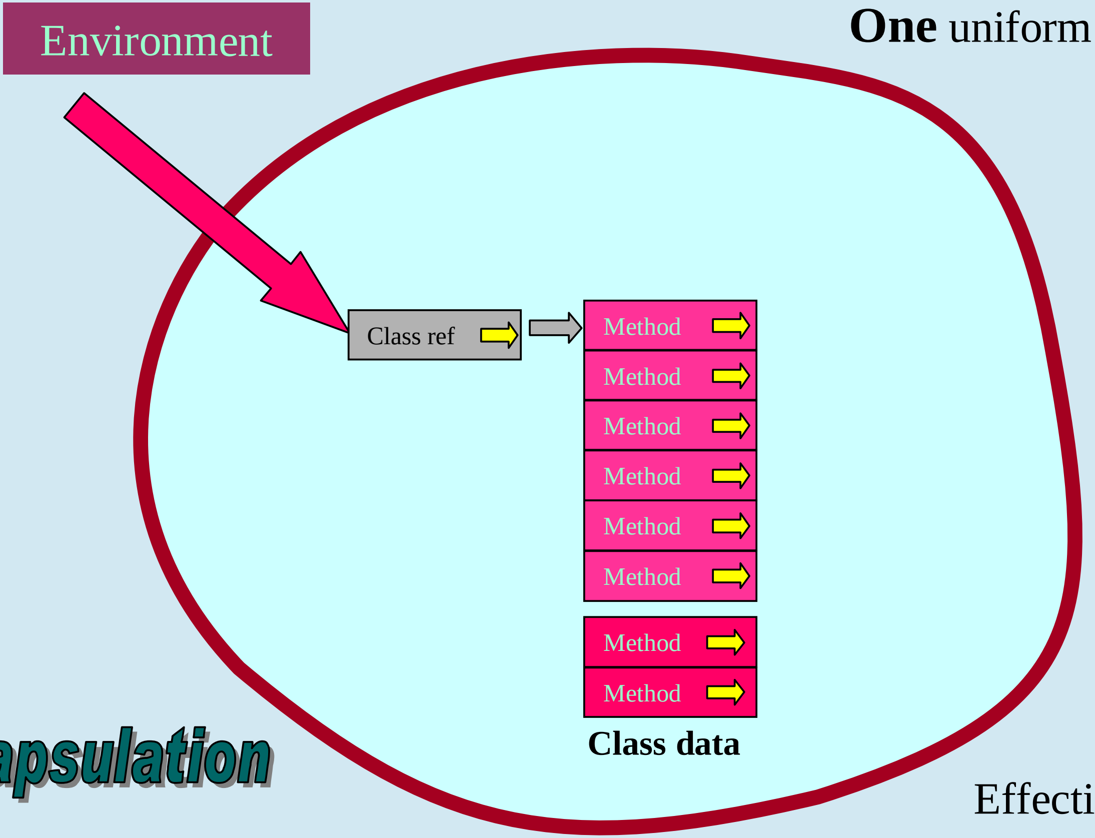
The individual encapsulates its own private state and the association with its class. Via this relation it also encapsulates its class.

The environment has an association with the class. When the individuals can be created dynamically, then this relation is used to command the creation of a new instance. The creation operation returns a reference to the new individual.

The associations of the environment to an individual are used to pass messages and commands to that individual or its class. Each of these associations corresponds to one of the interfaces of the individual. The Individual passes this information to its class and adds a reference to its own state. Dynamic instances can get a command to delete themselves.

OO → Component Orientation

One uniform access method



ADT++

Encapsulation

Effective **IPR** hiding

Speaker Notes

Most object oriented languages are careless with encapsulation. Often direct access to attributes is tolerated. This inhibits establishment of uniform access and makes it impossible to make the class instance fully responsible for its behavior.

Hard encapsulation may cure this flaw. Hard encapsulation enforces a single uniform communication protocol. This communication protocol may become scalable by extra services that are offered by the supporting infrastructure.

The Communication Protocol

- The choice of communication protocols must be limited to **a single scalable protocol**
- Because:
  - **All individual components must be able to communicate with each other**
  - **Even the simplest component must implement all existing protocols**

Speaker Notes

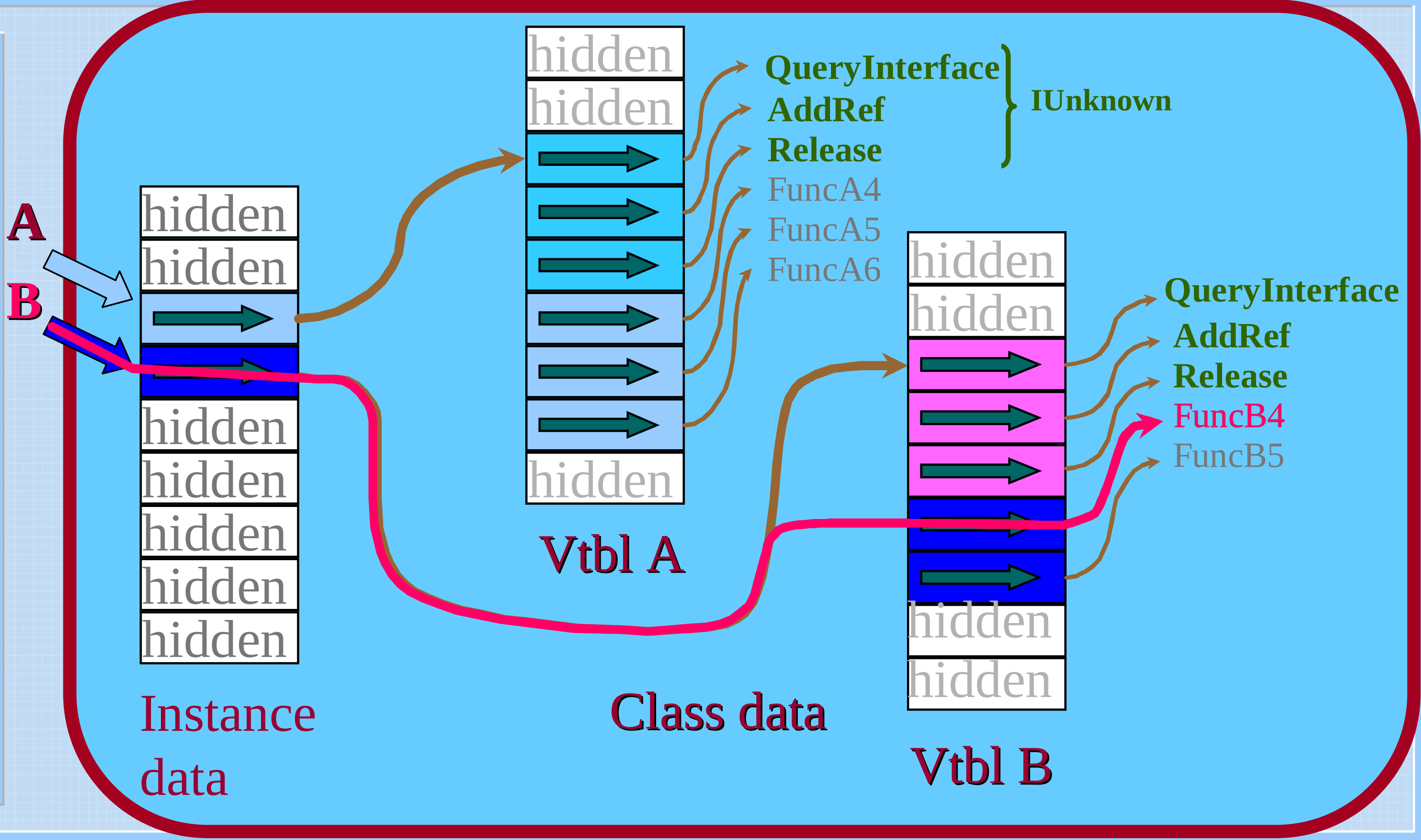
If several different communication protocols are accepted, then extra resources must be spent to protocol conversion and to manage groups of components that can handle a chosen protocol.

Scalable protocol means:

- Direct link at the lowest scale
- Dynamic link and/or script interpretation at medium scale
- Distributed access at the highest scale

Both the lowest scale of communication and at least part of the the script interpretation must be implemented by the component. Higher level services are implemented by the supporting infrastructure. In this way the corresponding resource needs are so small that they become acceptable for even the simplest component.

Object Interface (COM)



Speaker Notes

COM has a particular implementation for its binary structure. The second data structure, which represents the implementation of the class, is split into a series of subsets, called interfaces. For each of the interfaces the first data structure, which represents the instances of the class, contains a reference that points to that interface. Each of the interfaces contains pointers to three special routines:

‘QueryInterface’ supports maneuvering between interfaces.

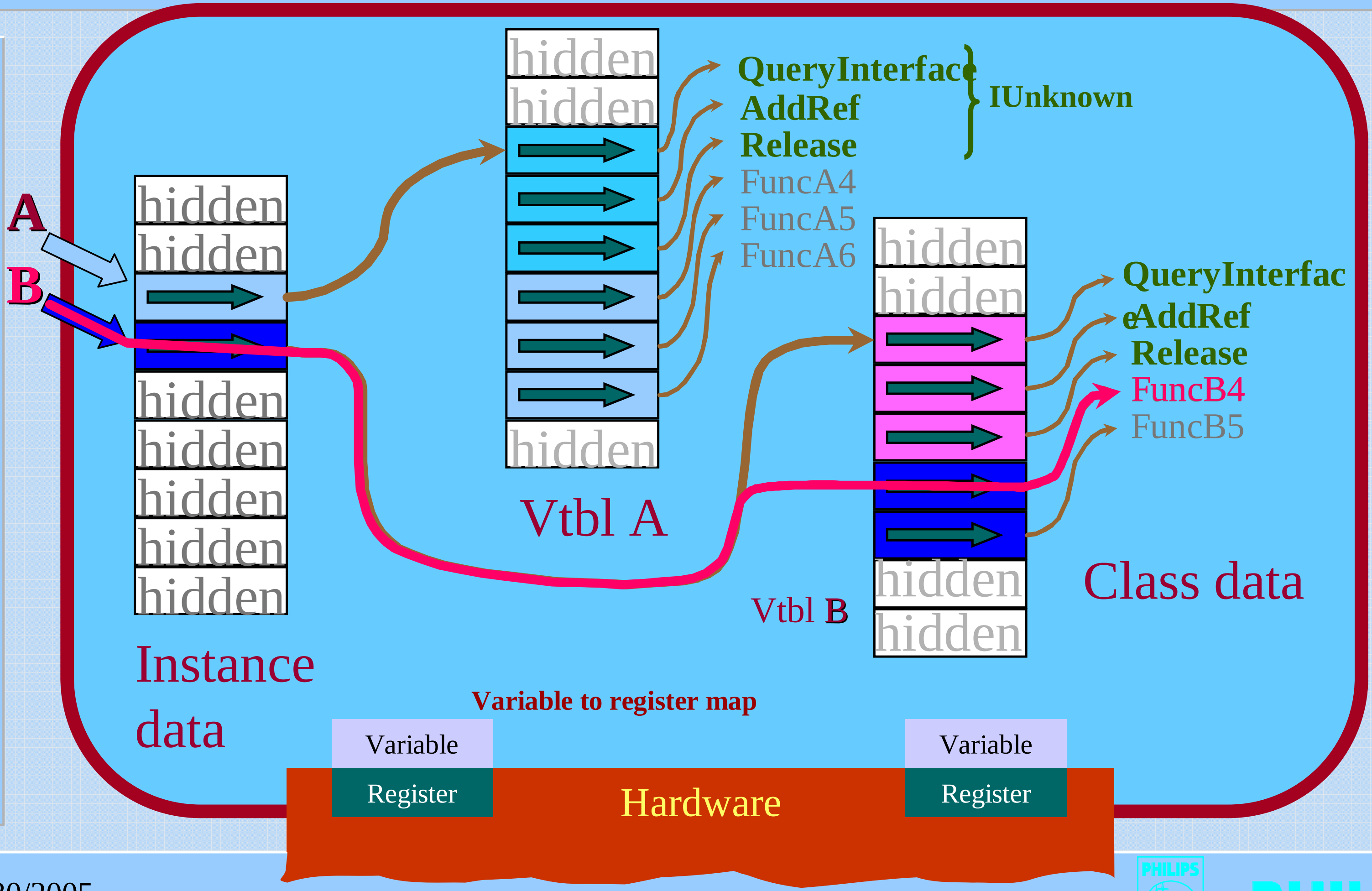
‘AddRef’ controls access to the interface. It increases a reference counter.

‘Release’ releases the access rights claimed via the ‘Addref’ call. It decreases the reference counter. It may delete the software component when the counter reaches zero.

‘QueryInterface’ uses globally unique identifiers (GUID’s) to identify the interfaces.

QueryInterface, AddRef and Release together constitute the IUnknown interface.

Object HWInterface (COM€)



9/30/2005

☺

PHILIPS

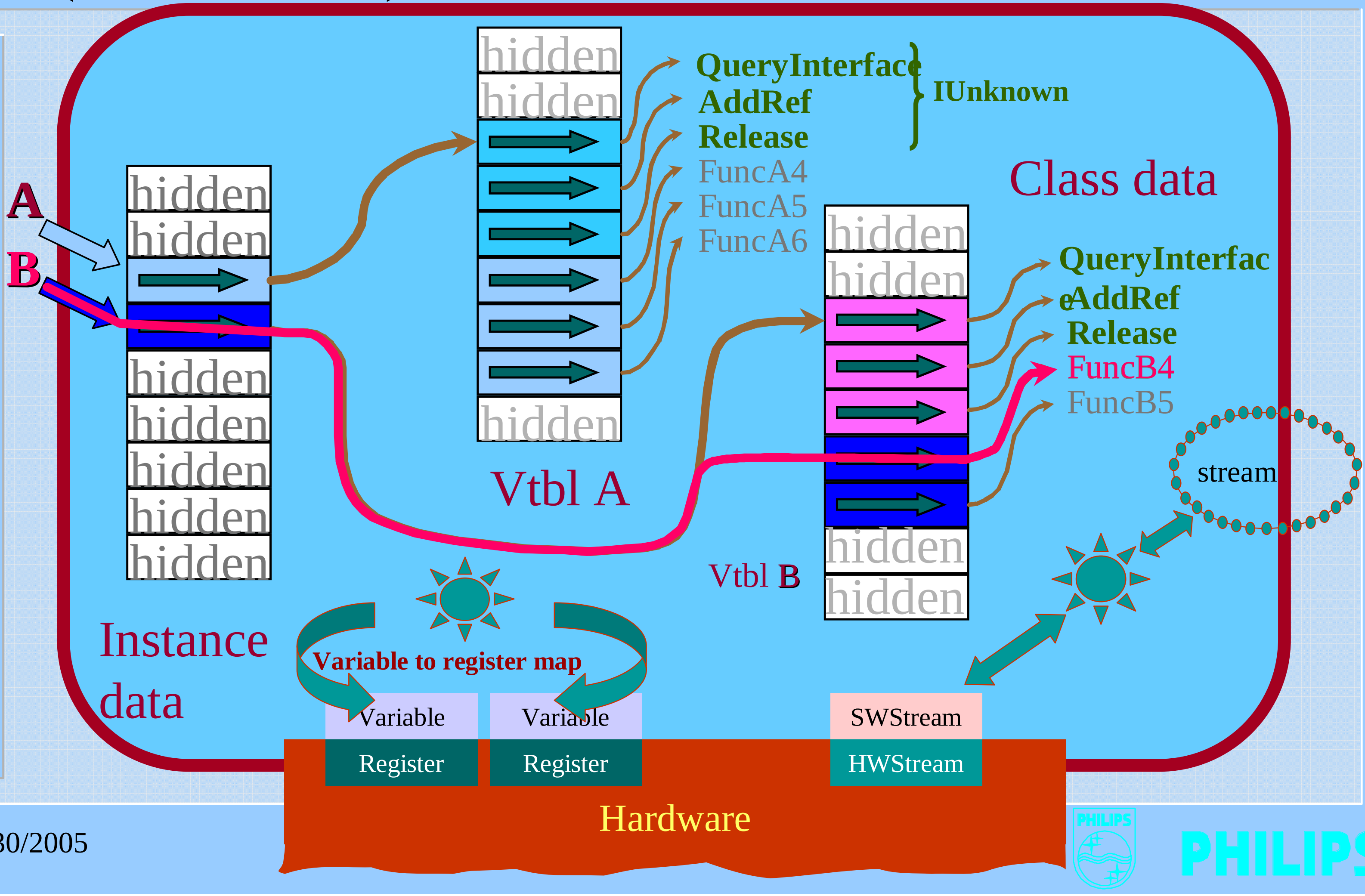
Speaker Notes

Apart from software/software interfaces components may also have have software/hardware interfaces. The interface definition is a mapping from a software variable to the bits of one or more hardware registers. In this mapping also the access protocol characteristics are described.

The map specification can be used to generate low level driver software and it can be used to generate VHDL.

The routines that are accessible via the software/software interfaces can (if they want) access the variables that are specified in the software/hardware interface.

Object Streaming Interfaces (COM€$)



Speaker Notes

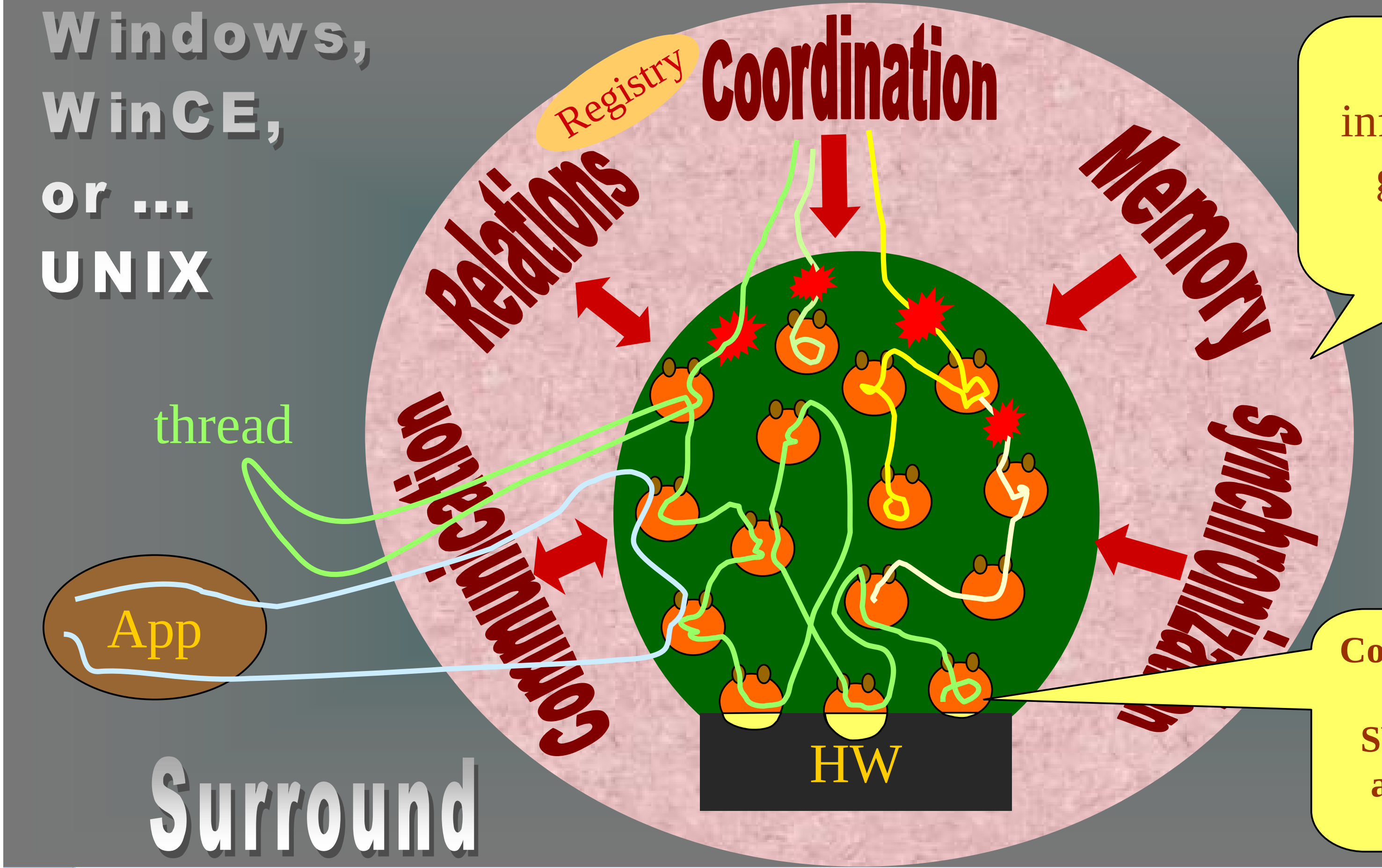
Apart from software/software interfaces software/hardware interfaces components may also have have SW/SW and SW/HW streaming interfaces.

The map specification can be used to generate low level driver software and it can be used to generate VHDL.

The routines that are accessible via the software/software interfaces can (if they want) access the variables that are specified in the software/hardware interface and the streaming interfaces.

Component based

Windows,
WinCE,
or …
UNIX



Tailored infrastructure generated by tools

Components can have SW interfaces and HW/SW interfaces

Surround

9/30/2005

PHILIPS ☺

Speaker Notes

The open (sub-)infrastructure provides inter-communication, relation management, memory management, task coordination and synchronization support to the software components. A sub-infrastructure may use and encapsulate these services from its surrounding infrastructure.

This tailored infrastructure is generated by the integrated design and build tool. It encapsulates the RTKOS. It is possible that the tool also generates a tailored RTKOS as integral part of the generated supporting infrastructure.

Both the relation manager and the task coordination makes use of a registry. This is a database where references to classes, instances and tasks are registered.

Speaker Notes

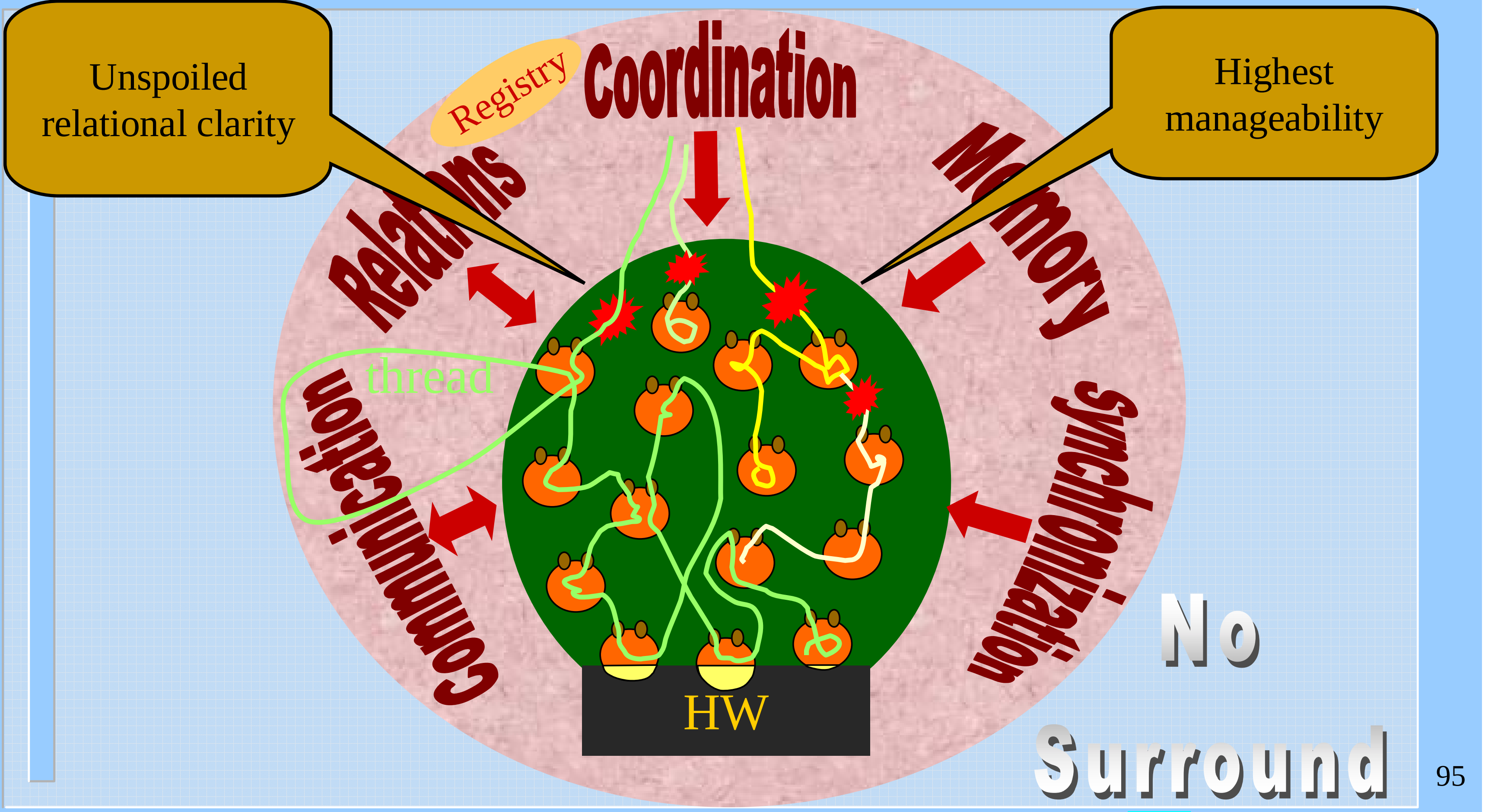
The task manager creates and starts tasks after system initialization. But tasks can also be started from running tasks. A task is started from a start routine, which is outside of the components. A start routine is called from the task manager (after initialization), or from a component, which is running another task.

Components can be called from tasks running in other components or from tasks that were running in the surrounding system.

Different tasks may start by letting their start routine call the same method of different (static) instances of the same class of SW components. It is also possible that a task start routine first instantiates an instance of the SW component class and then calls one of its methods.

In embedded applications software components will have one or more software interfaces and besides of that they also may have HW/SW interfaces.

No encapsulating system



Unspoiled relational clarity

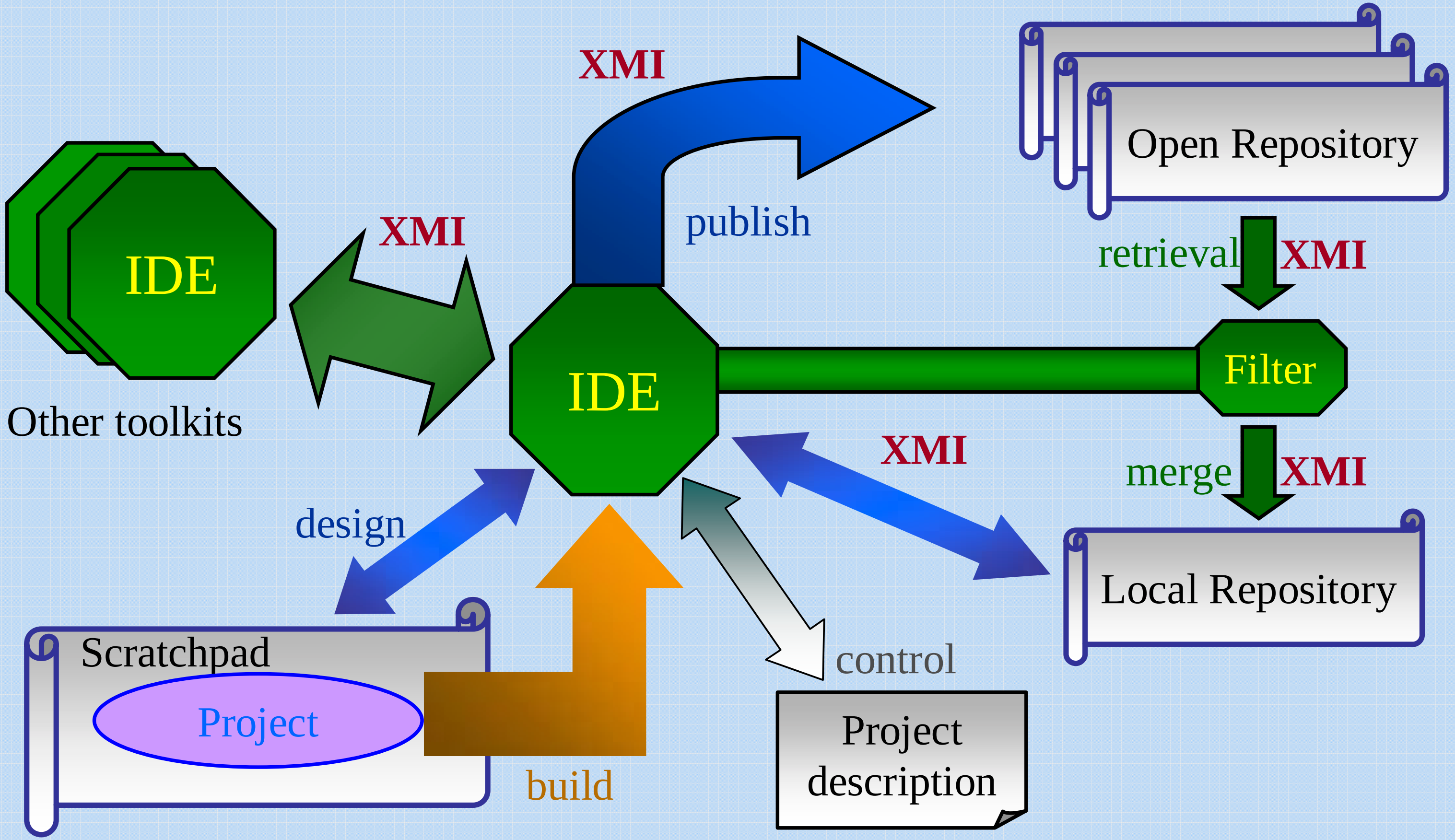
Highest manageability

No Surround

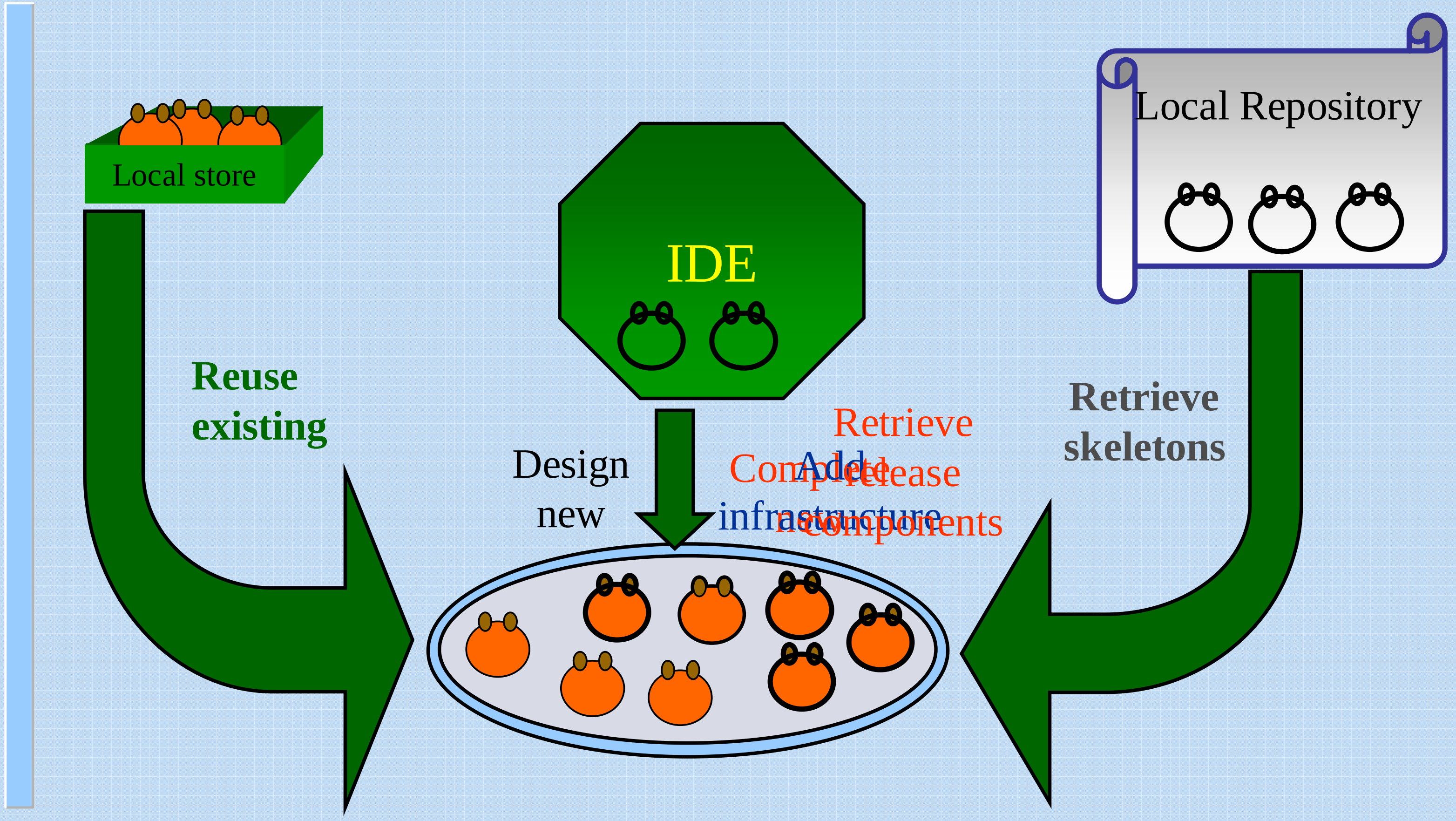
Speaker Notes

When the set of components is served by an infrastructure that is also constructed from components, and when that system is not encapsulated in a non-component based surround, then the effect of the healthy influence of component technology on relational clarity and as a consequence on manageability becomes most apparent.

Exchanging Design Elements



System configuration


Local store
IDE
Local Repository
Reuse existing
Design new
Retrieve release
Add infrastructure
Complete new components
Retrieve skeletons