Objects, Components, and Frameworks with UML

*The **Catalysis**™ Approach*

Desmond D'Souza

ICON Computing, a Platinum Company

Object, Components, Frameworks: Methods, Consulting, Training

"Looking for a few good minds!"

http://www.iconcomp.com

1

Welcome!

What is Catalysis?

Catalysis is a standards-based methodology for the systematic development of object and component-based systems. In this presentation, we cover highlights of the method and outline how it enables the development of models and designs from frameworks.

Catalysis has been adopted by several major projects, is being evaluated as the UML-based method and process for many more projects, has forthcoming tool support from multiple major tool vendors, and has services and related support available today.

Who is ICON?

ICON Computing was instrumental in influencing the UML (Unified Modeling Language) effort in 1996-97, bringing its work on *Catalysis* to bear on the emerging standards in the modeling market. ICON provides expert consulting and training in object technology and associated component-based development and frameworks. We offer skills in strategic planning, management of object and component projects, re-use, software architecture, analysis, design patterns, domain-specific patterns and frameworks, design, and programming. We emphasize good design and clear models in our courses, and transfer these skills to projects in our consulting and *HeadStart* programs.

Contact us at:

(512) 258-8437 Fax: (512) 258-0086

info@iconcomp.com http://www.iconcomp.com

What is *Catalysis*™?

UML partner, OMG standards, TI/MS standards

Precise models and systematic process

Dynamic Architectures

- A *next-generation* *standards-aligned* method
  - For *open distributed object systems*
    - from *components* and *frameworks*
    - that reflect and support *an adaptive enterprise*

From business to code

Compose pre-built interfaces, models, specs, implementations...

…all built for extensibility

Authors: D. D'Souza and A. Wills

Addison Wesley, *"Objects, Components, Frameworks…" 1997*

2

Taking apart the marketing jargon on this slide, we have:

next-generation: Catalysis provides a systematic process for the construction of precise models starting from requirements, for maintaining those models, for re-factoring them and extracting patterns, and for reverse-engineering from detailed description to abstract models.

standards-aligned: ICON has been a central player in the development of Catalysis and its contribution to standards including the Unified Modeling Language (UML), accepted as a standard by the Object Management Group in Sept '97, and component-specification standards defined by Texas Instruments and Microsoft, the CBD-96 standards from TI/Sterling, and the Cool:Cubes tool family from Sterling.

*open distributed object systems:*Our ultimate goal is to support the modeling and construction of open distributed systems -- systems whose form and function evolves over time, as components and services are added and removed from it.

*components:*Little, if any, modeling work should be done from scratch. If you draw two boxes and a line between them, chances are someone has done something very similar before, with an intent that is also very similar, if you only abstract away certain specifics. All work done in Catalysis can be based on composition of existing components, at the level of code, design patterns and architectures, and even requirements specification.

frameworks: In particular, some of these components are built so they are easily adaptable and extensible. We call these components *"frameworks"*, generalizing somewhat the traditional definition of a framework as a collection of collaborating abstract classes.

adaptive enterprise: And, we want to use these techniques from business to code and back.

Catalysis was originally developed by Desmond D'Souza and Alan Wills. The text on Catalysis was published by Addison Wesley in October 1998.

Outline

- **Method Overview - UML constructs with *Catalysis***
  - **Three Levels**
  - **Three Constructs**
  - **Three Principles**
- Component Specification - Types
- Component Design - Collaborations
- Component Architectures
- Refinements
- Business Example and Development Process
- Frameworks

3

Here is an outline of this presentation.

Method overview: Catalysis has a simple core, covering three levels of description, using three basic modeling constructs, applying three underlying principles.

Following this, we will examine the modeling constructs in turn, and then see how they fit together in the Catalysis development process.

Component Specification - Types: A *type* provides an external description of the behavior of a component or object.

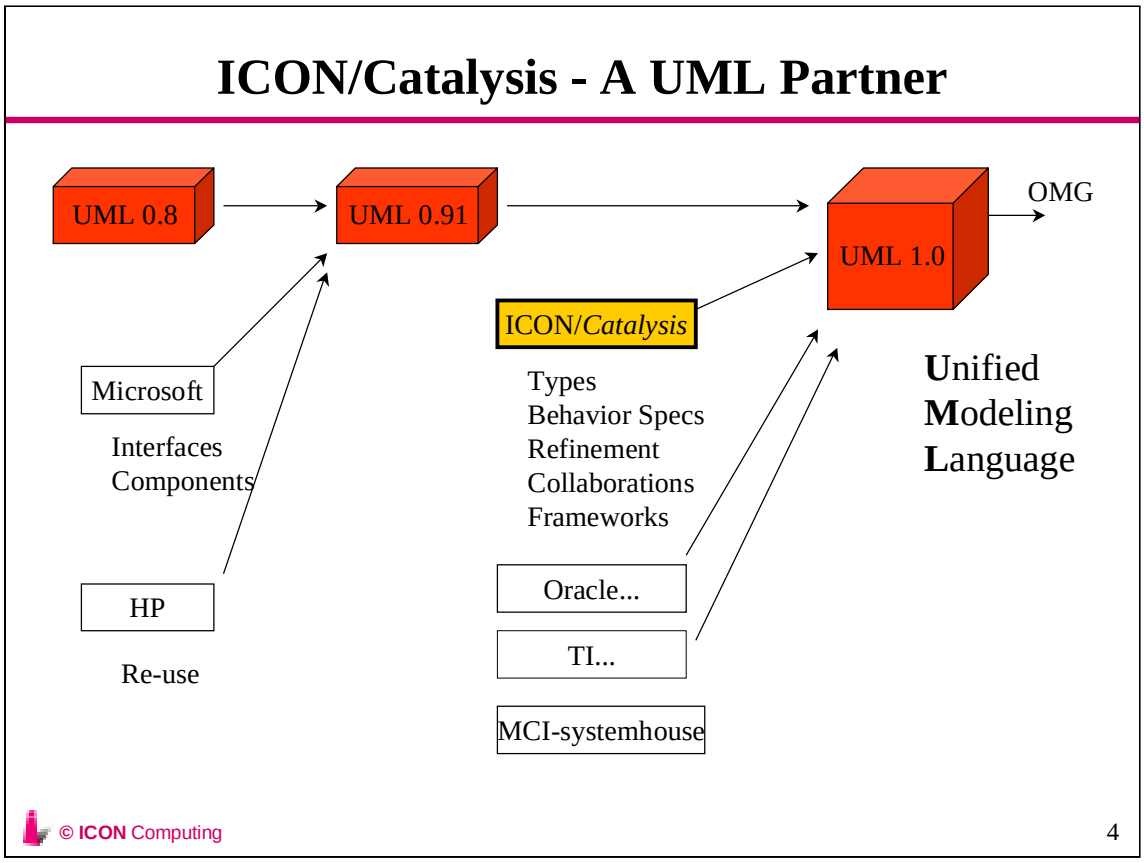
Component Design - Collaborations: A *collaboration* describes how the internal parts of a component interact to provide its required behaviors i.e. its internal design.

Component Architectures: Designing interacting components places some unique demands, and offers new opportunities, for architectural elements.

Refinement: One of the unique features of Catalysis is its support for multiple levels of description of the same phenomenon, based upon a strong notion of *refinement*. Refinement provides a well-founded basis for a *use-case driven* approach, and for the clear separation of external behavior specs from implementation.

Business Example: This section shows how Catalysis constructs can be used at the level of domain or business models.

Frameworks: And lastly, we illustrate the power of the *framework* construct in Catalysis. Frameworks capture common patterns of interactions, requirements, design transformations, and more, in a form that is both abstract enough to permit adaptation and application in many different contexts, and precise enough to permit sophisticated tool support and reasoning.



First, a brief bit of perspective on the role Catalysis has played in the UML.

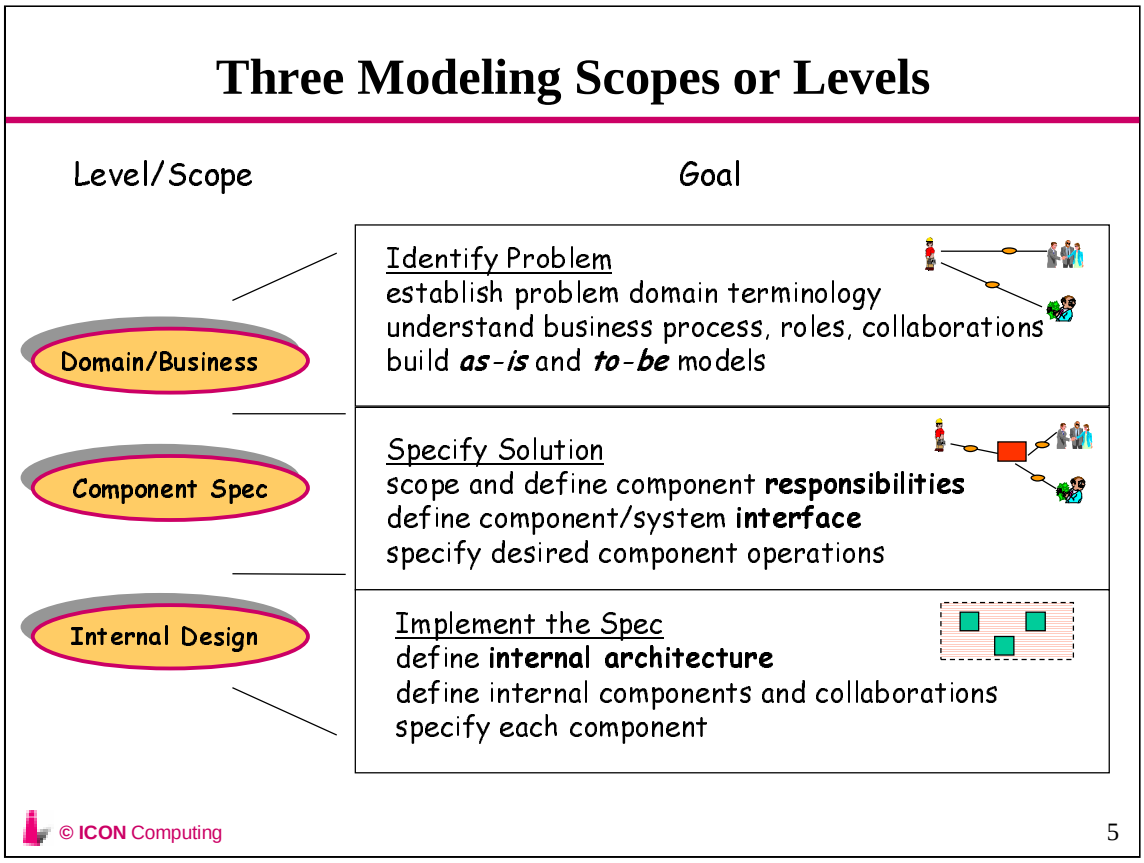
UML 0.8 was released by Rational in October 1995, based on the work of Booch, Rumbaugh, and Jacobsen.

In Fall 1996, Microsoft and Hewlett-Packard joined the consortium of companies that was defining UML, initiated by Rational. Microsoft brought some modeling requirements based on its COM model of interfaces of components; HP brought a focus on re-use.

In Oct-Nov 1996, several other companies joined this consortium. Oracle came in with a particular interest in the modeling of business processes and workflows. Texas Instruments software came in with an interest in components and business modeling, and in Catalysis. MCI Systemhouse's interest was distributed systems.

And **ICON Computing** joined this lofty team with its work on Catalysis. As a partner in defining the UML 1.0, ICON brought a focus on several key elements of Catalysis: types (as distinct from classes), precise specification of behavior, collaborations as definitions of multi-party behavior, refinement of models, and frameworks as a means to abstract and apply recurring patterns in modeling. This technical contribution to the UML team was largely done by Desmond D'Souza, with input from his Catalysis co-author, Alan Wills.

The UML 1.0 spec was submitted to the Object Management Group in Jan 1997, and accepted as a standard in September 1997.



Catalysis supports three levels of description, which, recursively, span most interesting levels of modeling problems.

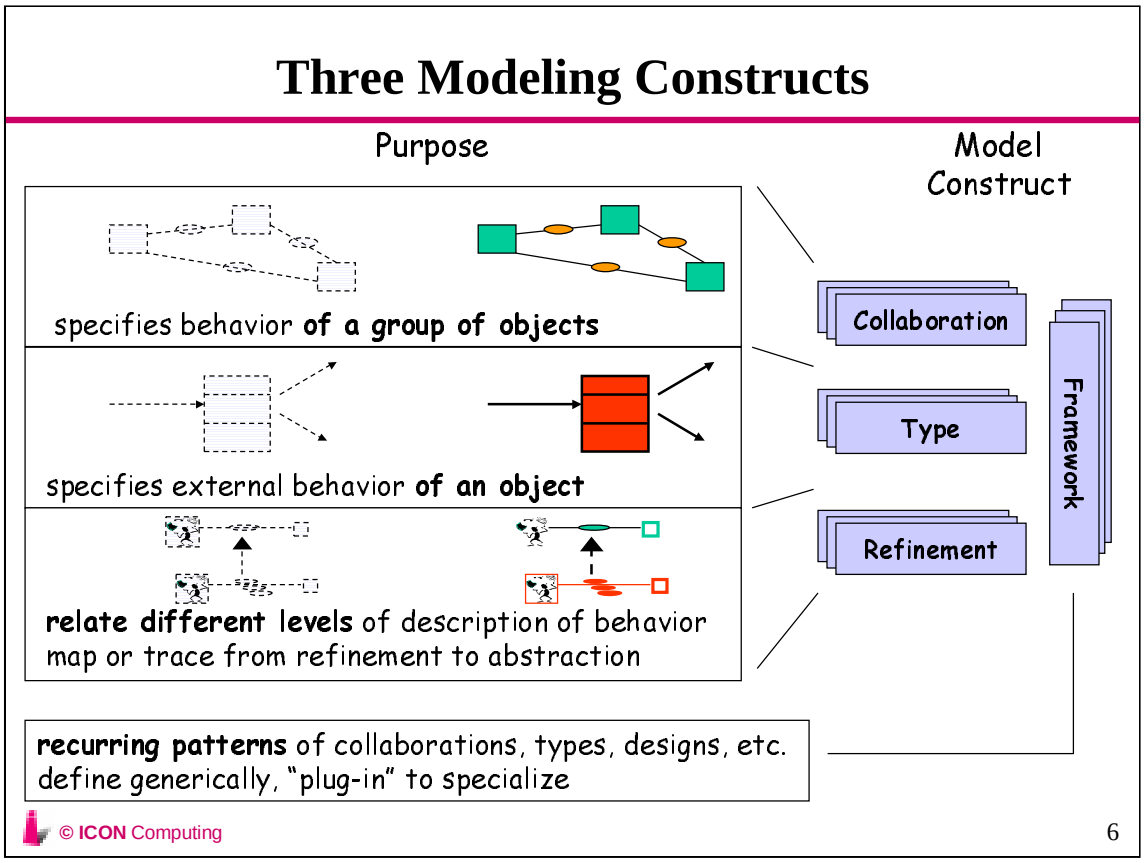
Domain/Business Models: For any component, a domain model describes the context in which that component will reside. This may be a model of the environment for a real-time system, or a model of a business for an information system or business application, or a model of terms and concepts for some other application. Domain models can have an *"as-is"* element distinct from a *"to-be"*. This is particularly true in the case of business systems, or the re-engineering of an existing system, where the as-is model will be modified to look like the to-be model, requiring some changes in the environment or context itself as well. For example, we may decide to introduce a new component into a business, and change some of our business processes accordingly.

Domain models can be interesting and useful independent of any software system or component. In these cases the domain model describes some phenomenon or subject area that is deemed of interest for some purpose.

Component External Specs: This level focuses on the component(s) of interest -- usually those to be built or modified -- and builds a description of its externally visible behavior with its environment.

Component Internal Design: This level effectively *"opens the box"* for a given component, and describes how it is designed internally to provide its externally specified behavior. It describes the different implemented parts that comprise it, their connections, and their interactions.

Note that we can, at this point, specify any one of these internal components in terms of its externally visible behavior *in the context of the component internal design ,which constitutes* ***its*** *"environment"*. This process is recursive as far as necessary or useful. We would typically stop at the level of components that exist, or that can be bought or built.



Catalysis has two primary modeling constructs:

Collaboration: A description of how a group of objects jointly behaves when configured together in specific ways.

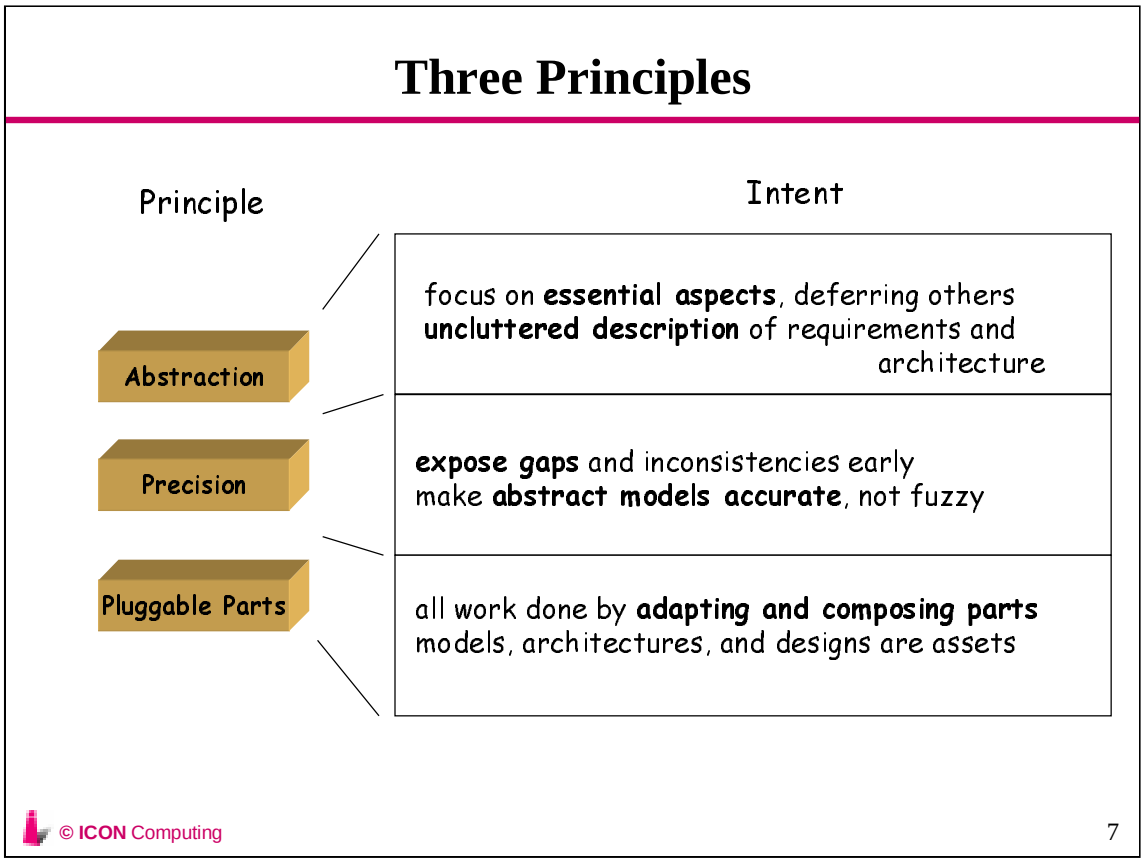
Type: a specification of the externally visible behavior of some object, abstracting from details of its internal representation, algorithms, and data-structures. A type is, in fact, a degenerate collaboration; it says as little as possible about objects collaborating with the one under focus.

For either a individual object or a group collaboration, we can model the description at different levels of abstraction. For example, a collaboration may be described in terms of very abstract actions between the participants, or in terms of a detailed point-to-point protocol of individual messages. *"You attend this seminar"* is an abstract description, whose refinement could be *"You sign up for the seminar; listen and ask questions, and provide feedback and criticism"*.

Refinement: A relationship between two description of the same phenomenon at two levels of abstraction. To justify the claim that one is a refinement of the other, there must be some *mapping* to each element of the abstract description, from some elements of the more concrete one.

Across many types, collaborations, or refinements, we will encounter recurring patterns of similar structures of designs or specifications of structure and behavior. For example, the *collaborations* involved in submitting an employment application to a company are similar to the collaborations involved in submitting a printing job to a printer. Similarly, the specification for allocating slots of machine time to production lots on a factory floor is similar to that for assigning vehicles to rental requests at a car-rental company. And, the *refinement* involved in making an ATM user validate their card before withdrawing money is similar to the refinement in making a user logon before using a computer system.

Framework: Captures a pattern of types, collaborations, refinements, design transformations, etc.



No method can really be coherent with some underlying principles that it (or its creators) hold dear. In Catalysis, these principles are:

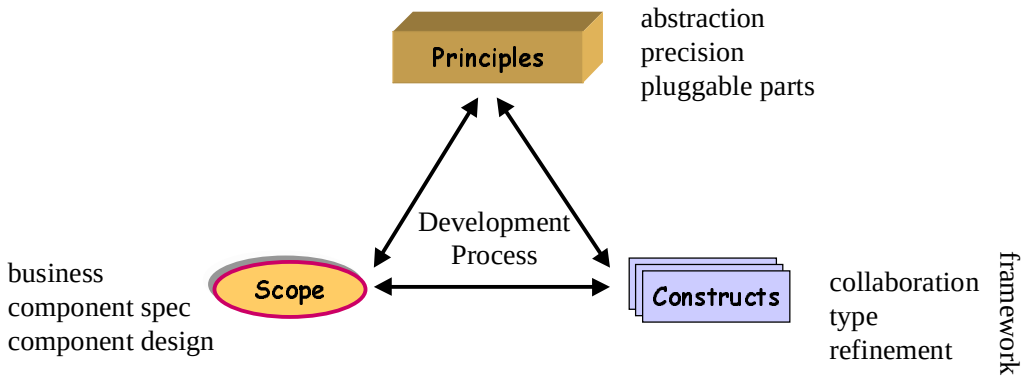
Abstraction: It is overwhelming to deal with the complexity of a software system at the level of code. Abstract models of that code let us separately describe its functional behavior, its relationship to the problem domain, its performance characteristics, the architecture it uses, etc. Any piece of source code will have been written to satisfy a combination of requirements imposed from each of these different points of view, each describing a different aspect of the problem. Examples of common abstractions are *interfaces* (which hide information about *implementation*), and *architectural patterns* (which hide details of usages of those patterns in different domains).

Precision: Abstract descriptions, unfortunately, have a history of being extremely fuzzy and poorly defined. As a consequence, a very elegantly laid out drawing of boxes which claims to describe the system architecture may not have sufficiently precise meaning to show that a particular implementation either does or does not conform to that architecture. Abstraction does not require loss of precision -- provided the appropriate tools and constructs are used. Abstraction with precision lets us work at different levels of detail with some confidence that the abstractions are accurate representations of the eventual implementation.

Note, however, that precision must have its rightful place. There is always need for creativity, and formalism is not always appropriate. Different projects and development staff also justify different degrees of precision.

Pluggable parts: All development work should be done by (a) adapting and assembling existing parts, be they implementation code, requirements specs, or design patterns, or (b) disassembling existing design to re-factor out more abstract parts from it.

A Uniform Approach



- Constructs and principles apply (recursively) at all levels

The core of Catalysis is simple partly because the three legs it builds on - its constructs, principles, and levels of scope - are simple and consistent in the way they interact with each other. For example:

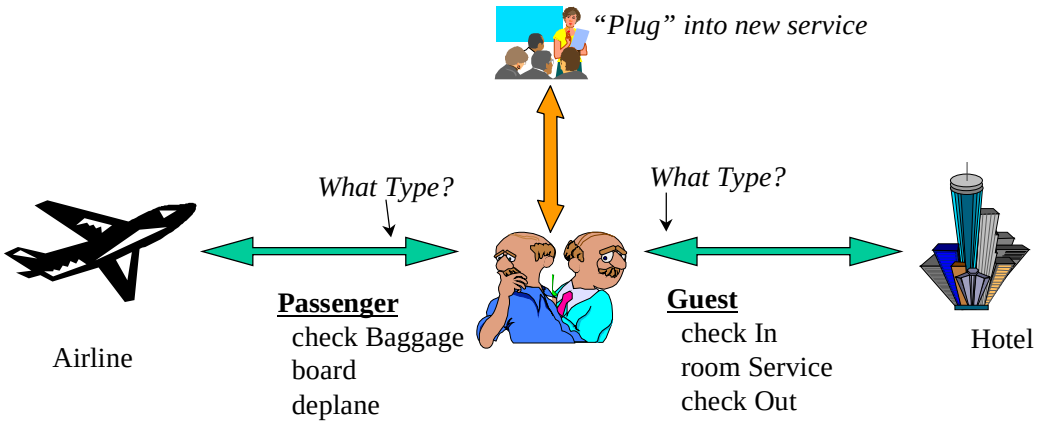
> In specifying a *business model*, we may build a *collaboration* to describe a business interaction. We keep this description *abstract* by deferring details of the protocols by which the parties interact and exchange information. We still want this description to be *precise*, however, and use our modeling tools to make it so. A more detailed level will *refine* the actions in this collaboration to a much more detailed interaction sequence. Because the abstract description was nonetheless precise, the refined model can be shown to conform it (or be refuted!).

The strength of the approach comes from the fact that the small number of *core* concepts form a *coherent* whole, and can be combined together in a great many different ways.

Outline

- Method Overview
- **Component Specification - Types**
- Component Design - Collaborations
- Component Architectures
- Refinements
- Business Example and Development Process
- Frameworks

Type of an Object: Multiple (Inter)Faces



- Object or component plays different roles
- It offers multiple interfaces, and is seen as different types
- Benefits to the *airline*, *hotel*: less coupling, more "pluggability"
- Benefits to *person*: easier to adapt to "plug" into a bigger system

One of the most fundamental aspects of component-based design is that of a component interface.

And, one of the first things we discover about component interfaces is that one component may offer multiple interfaces. Just as an object, in the real world, can play different roles, and hence be perceived as different types by different collaborating clients. For example:

A person, in his interactions with an airline, is perceived as being *of type Passenger*. Specifically, any object that could provide *passenger*-like behavior -- check baggage, board, deplane, etc -- could be "plugged into" an airline and travel. This is true regardless of details of how that person *implements* these behaviors.

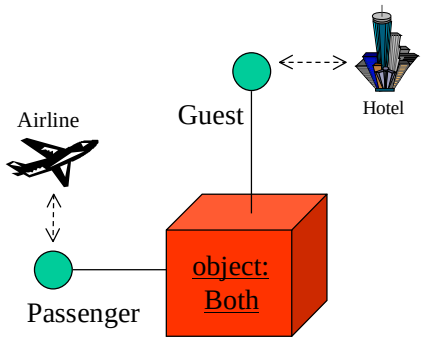
That same person, in his interactions with a hotel, is perceived as being *of type Guest*. Any object that could provide *guest*-like behavior -- checkin, room service, checkout, etc -- could be "plugged into" a hotel and enjoy a stay.

The *type* of a component -- a description of the interface it offers a client -- varies across its interfaces.

In addition, if we need to integrate this object or component into a larger system in which it needs to "plug-into" a new service, it will typically mean a new interface is required e.g. the person must be capable of playing a *Student* role to participate in a broader scope system.

Types: Java, COM, Corba, ...

```
interface Guest {
    checkIn ();
    roomService();
    checkOut ();
}

interface Passenger {
    checkBaggage ();
    board ();
    deplane ();
}
```



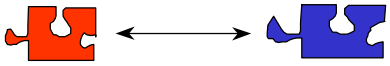

```
class Both
  implements Guest, Passenger {

    // implementation ….
}
```


Here is some Java code illustrating the point.

An **interface** is a distinct construct from a **class**, and a class can implement many interfaces.

Component

- There are many definitions of *component*
  - A user-interface widget that can be assembled on an interface builder
  - An executable module with a specified interface
  - A large-grained object with encapsulated persistent state and an interface
- All have the goal of *software assembly*

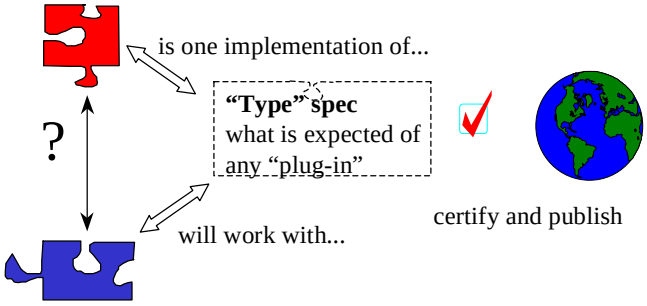


- An independently deliverable unit that encapsulates services behind a published interface and that can be composed with other components
  - Encompasses executable, module, and model/specification units

So, now that we know roughly what an interface is, what exactly is a component? Our definition, shown above, is more general than most.

Components *need* Precise Interfaces

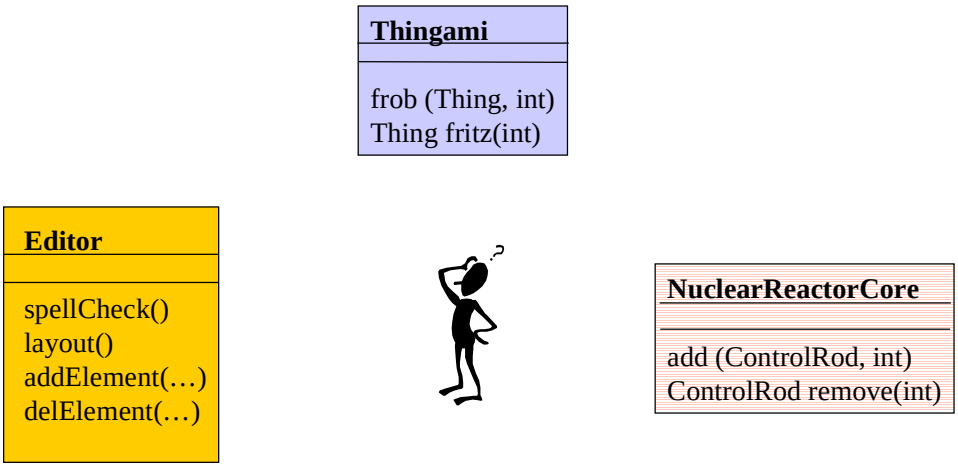
- To build systems by “plugging” together components



- Two choices: “plug-n-pray”, or use better specifications that are:
  - **abstract**: apply to any implementation
  - **precise**: accurately cover all necessary expectations

If we are to realize systems by “plugging” together components that may be specified and built by differently people in different parts of the world, we need clear specification of component interfaces. Otherwise we stand little chance of realizing robust systems by assembly.

The "Black-Box" Component



- Signatures are not enough to define widely-used components

Here are 3 different components. Would you use them as black-boxes?

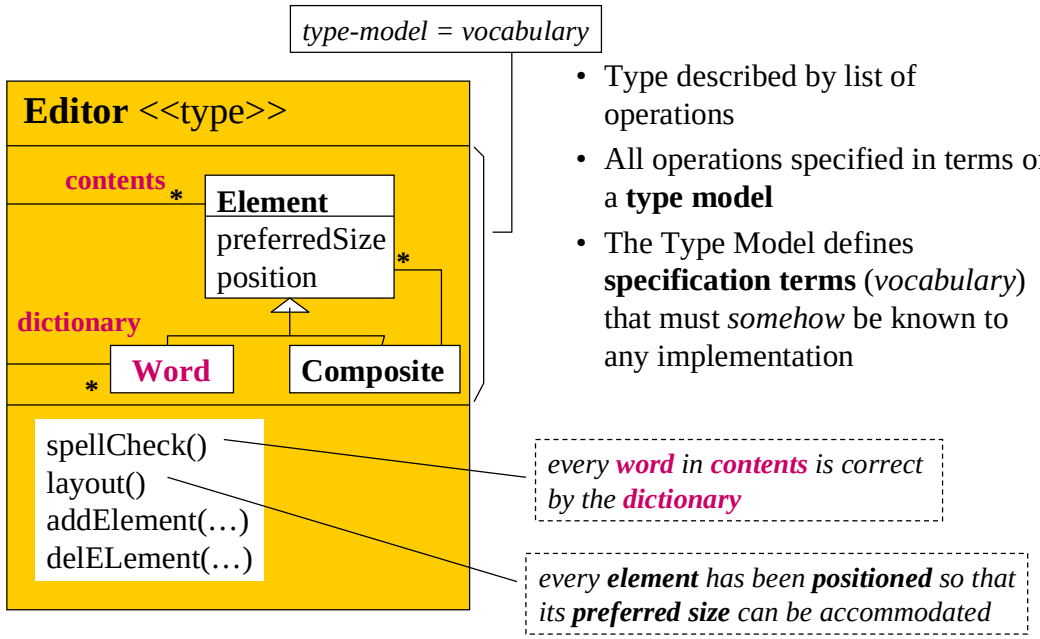
Thingami: probably not, because you have no idea what any of it means.

Editor: well, this one seems more likely. We understand the words, and know what they mean. SpellCheck corrects the spelling of words in the document, and layout probably figures out line and page breaks, and positions of floating figures and tables.

Nuclear Reactor Core: well, we know what this one does, but are not likely to touch it!

In hiding the implementation of a component, we cannot make do with describing the interface simply as a list of operation names. Instead, we must also provide some specification of the behavior guaranteed by those operations.

Model-Based Type Specification - Example



We start with an empty "type" box. The UML *stereotype* **<<type>>** indicates that this box is an external abstraction of any internal implementation. In particular, the middle section of the box, which is normally interpreted as stored data, is now interpreted as *abstract attributes*. In the model above, the editor has two interesting abstract attributes, *contents* and *dictionary* (that happen to be drawn as lines called *associations*). None of the elements in this section need be stored, or directly represented in any way.

Here is how we go about specifying the type Editor:

List the operations

Informally specify each operation

Identify the hidden terms or vocabulary

Model this vocabulary as attributes (or inputs, outputs)

Attributes can be drawn

Attributes have types, which have attributes, …

Re-formulate the operation specs in terms of this model

Validate any implementation by mapping first to this model, then reviewing the operation code

A given implementation need not directly store *contents* or *dictionary* e.g. we may store the dictionary as a reference to a file which is indexed by character prefixes. However, it would be hard to conceive of a correct implementation of this specification which had *nothing whatsoever* that corresponded to the words constituting the dictionary. In other words, every correct implementation of an editor must have something that can be *mapped to* the concepts of the words in the dictionary of that editor. Re-stated yet once more, the type model must be a *valid abstraction of* any correct implementation.

But, *What about Encapsulation*?

- Encapsulation is about hiding *implementation decisions*
- It does not make sense to hide *interfaces*
- And interfaces always imply *expected behavior* as well
  - This is what *polymorphism* is all about
- Hence, specify a ***minimal model*** of any implementation

What about polymorphism?? Are we not compromising that holy grail by describing attributes? Are we not giving away too much in what should have been an interface specification?

Remember some basic points:

Encapsulation is about hiding implementation decisions, not about refusing to tell the client about the behavior guarantees you do, and do not, make.

We already know that an interface specified merely as a list of operation names and signatures is really quite inadequate for a client's purpose. You have to describe behavior guarantees as well. The trick is to describe these guarantees in a way that tells a client enough, but still permits many different implementations. *This is what polymorphism is all about -- specifying visible behavior without compromising implementation alternatives.*

Hence, we specify a type based on a *minimal abstract model* of how any valid implementation.

Formalizing Operation Specs

```
Editor:: spellCheck ()
post    // every word in contents
        contents#forAll (w: Word |
        // has a matching entry in the dictionary
                dictionary#exist (dw: Word | dw.matches(w)))
```

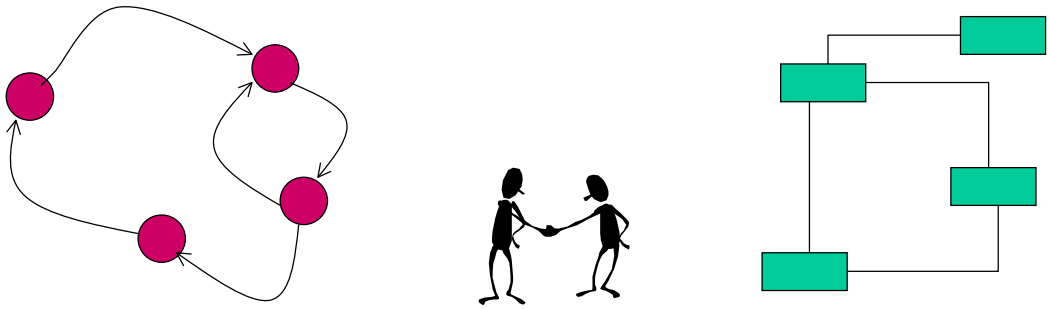
- Operation specification can formalized
  - *OCL* (Object Constraint Language) used after rectifying some shortcomings
  - Checked, used for refinements, testing, change propagation, …
- Enables checkable/testable design-by-contract

Operation specs can be made as precise as appropriate, even to the point of being machine checkable. While this may seem alien at the level of requirements -- where we are accustomed to being quite fuzzy about what we mean, and to avoiding tricky issues under the guise of "*oh, that's a design detail*" -- the discipline it enforces actually has very tangible benefits.

One very concrete benefit is that tests are specified as an incidental side effect of writing a precise specification of behavior. If, for any test case, this specified outcome did not result, then the test has failed the specification.

Of course, a given implementation may not have anything that directly represents *contents* and *dictionary*. However, as we said earlier, it must have something that can be *mapped* to these attributes, even if it takes a complex function to compute this mapping. If this mapping -- formally called an *abstraction function* or *retrieval* -- is written down in the code, then the above specification becomes an *executable test specification* as well.

Behavior vs. Data-Driven -- a Non-Issue



- A long-standing controversy: *"should models be data- or behavior-driven"*
- The approaches are resolved cleanly in Catalysis

- Behavior induces abstract attribute models
- Attributes support behavior specification

There has been a long-standing controversy about data-driven Vs. behavior driven methodologies. Certain methods have been characterized as behavior driven (CRC, Booch, OBA, etc.), while others have been characterized as data-driven (OMT, Shlaer&Mellor). Some object purists reject data-driven approaches on the basis that objects are primarily about responsibilities and behaviors, and about hiding implementations. On the other hand, data-modeling people find it hard to understand how considerations of data can be so thoroughly avoided (or deferred).

The Catalysis interpretation of attributes and "type-models" makes this controversy is largely irrelevant. The basic reasoning is actually very simple, and follows this progression:

Pure behavioral "black boxes" still need their behaviors to be specified

Given an abstract model of attributes, behaviors can be described precisely

Abstract attributes do not represent data storage, or even public methods; just precisely defined terms

Attributes can be drawn visually as "associations" in "type models"

Abstract attributes can be parameterized

The type model provides a precise vocabulary for defining behavior contracts

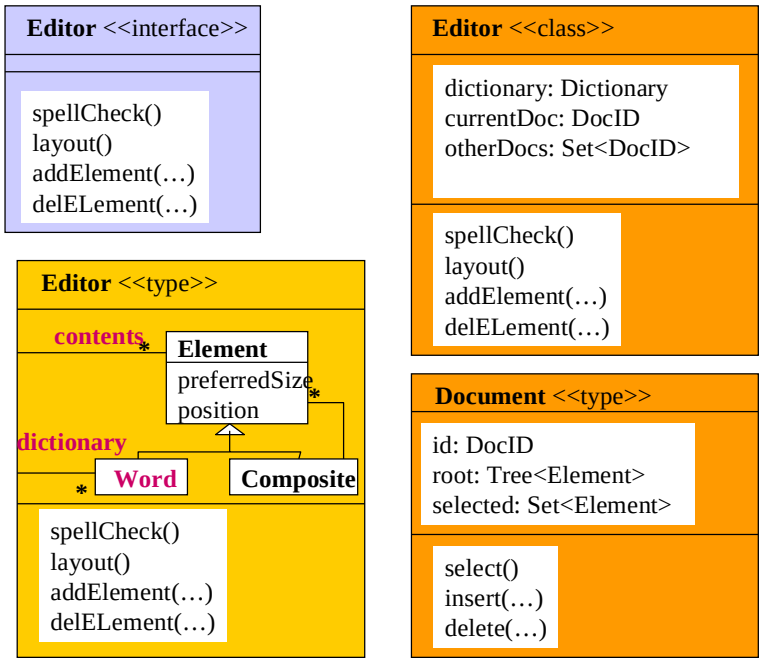
Any collaboration requires mutual models

Models from multiple "views" may be composed

Collaborations may be refined in mutually interesting ways

Details on this are available on http://www.iconcomp.com/catalysis under the topic "Behavior- vs. Data-Driven Methods".

But, *Which Models Are "Right"?*



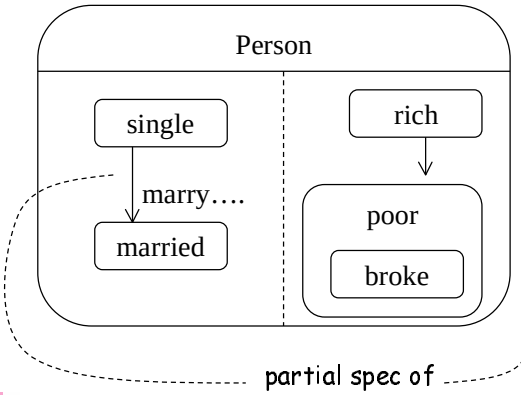
- Type model abstracts several implementations
- Each model element has some mapping from an implementation

Of course, there are many possible type models that can support specifications of equivalent behavior. In fact, the very fact that behavior can be specified in terms of an abstract model, and yet written in terms of a potentially different implementation representation, requires this.

A good development process will prefer type models -- which are, after all, about external specification of behavior -- that are based on concepts and terminology from the domain or business at hand.

Attributes and State Models

- A state is a boolean attribute
  - Attributes do not have to correspond to stored data
- State structure defines invariants
  - Sub-states and “concurrent” sub-states
- Transitions define (partial) action specs



Person

single, married: Boolean
rich, poor, broke: Boolean
inv single **xor** married
inv rich **xor** poor
inv broke **implies** poor

spouse: Person
inv married **iff** spouse <> nil

marry (other: Person)
pre single & other.sex…
post married & spouse = other & ...

States are not a additional construct in Catalysis. Instead, a state corresponds to a boolean attribute: an object either *is*, or *is not*, in a given state at any time. Recall that attributes do not have to correspond to stored data, or to implemented public (or even private) methods, so long as there is some mapping from the implementation to each attribute.

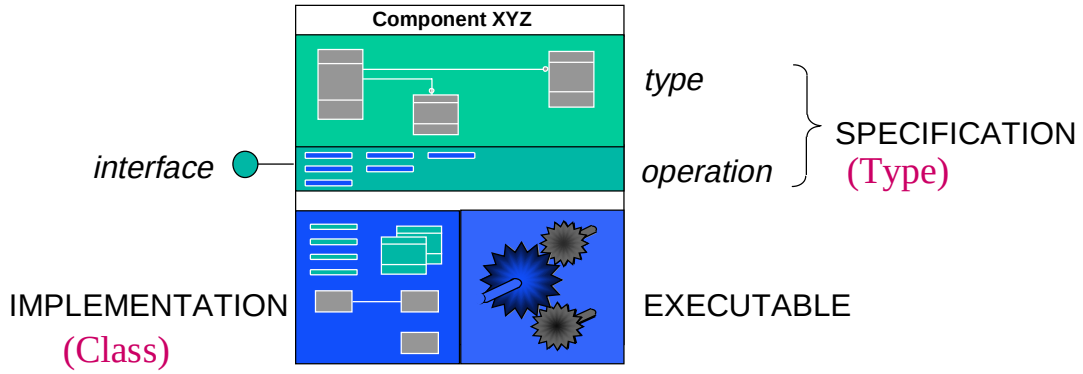
The structure of states on a state-diagram visually defines invariants on these attributes. Thus, separately drawn states must be exclusive; nested states correspond to boolean implication: *sub=>super*.

There are usually additional invariants between “state” attributes and other attributes and associations. For example, a person is in the *married* state if (and only if) he/she has a spouse.

State transitions drawn on a state-diagram define (partial) specifications of operations or actions. The complete specification of any action is defined by some composition of all transitions that name that action as their trigger.

This gives state transition diagrams the same basis as all other modeling constructs in Catalysis, and enables better integration and consistency checking.

Component Models: Catalysis, TI, M.S. Repository

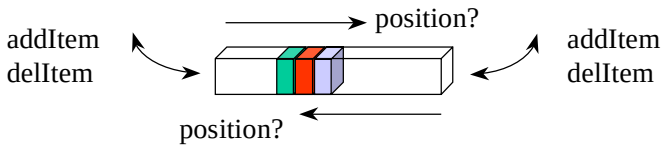


- Every component description consists of at least 2 parts
  - The implementation (designed classes, modules, exe, etc.)
  - The specification of the component interface(s) as Type(s)

Any packaging of a re-usable component must include not just the executable code, but also an interface specification. Catalysis has influenced the component-description standards being used by Microsoft and TI Software (now Sterling Software), through a collaboration between ICON Computing and Texas Instruments.

Java Interfaces: which behaviors are OK?

```
interface awt.ListBox {       // not exactly awt.List
   // represents a list of items to be offered on a GUI
   void addItem (Item, int);
   Item delItem (int);
   …
}
```



- Only some implementations are valid
  - The signature-based interfaces does not specify which

Consider a simple Java interface shown above. The signatures alone do not adequately specify the required behavior e.g. are items positions counted from the front or back? Starting at 0 or 1? What happens if I add at a position beyond the current contents? What does add do to the item already at a position, if any?

A common vocabulary relates operations

- Specification of operations uncovers a *hidden vocabulary*
  - addItem (item, pos)
    pos must be between 1 and *current count* + 1
    the item has been inserted at position and is *selected*
    all previous items from pos to current count *moved up* by 1
  - delItem (pos)
    pos must be between 1 and *current count*
    the previous item at pos has been returned
    all previous items from pos+1 to current count *moved down* by 1

As we write down a description of the effect of each operation we will find ourselves introducing a *hidden vocabulary* i.e. a set of terms which are needed to describe the operation, but which are not themselves a part of the run-time interface. As these terms are identified, we can start to define them more precisely, and to use the same terms to uncover questions and describe other operations as well. Here is one possible discussion:

A: At what positions can I insert an item?

B: Any positions in the current range should work fine.

A: And what would that current range be?

B: It depends on how many items have been added to the list. You can insert anywhere from the start to the end of that list.

A: So we could insert at position 1 upto (including) one past the last element. Could I describe that with the *current count* of the list?

B: I suppose so. The count changes as you add or remove items.

A: What happens to the item which was already at that position?

B: Oh, it is moved up by one (along with all other items)

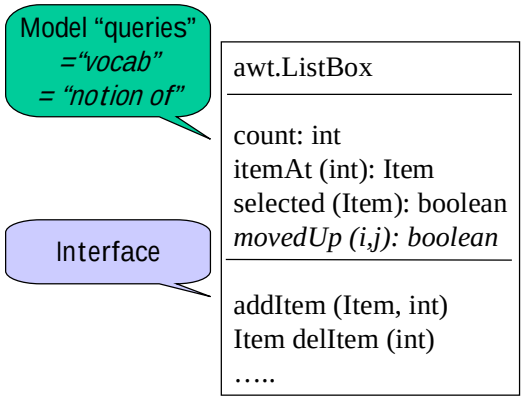
A: OK, so we can use *move up* and *move down* to help describe these operations?

.....

And so on, introducing new terms only when needed to describe operations.

Model vocabulary formalized as “queries”

```
interface ListBox {
    /* pseudo-java
    model {
        int count();
        Item itemAt (int pos);
        boolean selected (Item);
        movedUp (i,j)
    } */

    void addItem (Item item, int pos);
    Item delItem (int position);
    …..
}
```



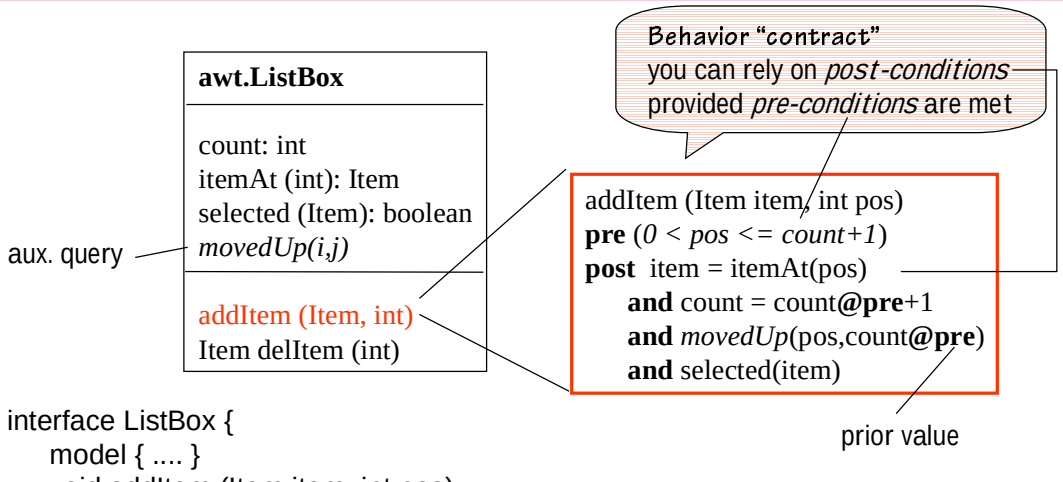
Note: The model “queries” do not represent an implementation!

Some terms can be defined in terms of others

e.g. *movedUp* can be defined in terms of *itemAt(I)*

The *“vocabulary”* used in specifying operations can be made formal using type-model attributes. Such an attribute is best interpreted as precisely defined *terms*, or a *hypothesized query*. It does not need to be stored directly, or even be publicly accessible, in an implementation. However, the client of a type, and the implementor of that type, had better both agree to the definition of that term, and to how the publicly visible operations are related to the value of that term.

Type = Behaviors specified using Model




```
interface ListBox {
    model { .... }
    void addItem (Item item, int pos)
      pre   0 < pos and pos <= count+1 // provided position is in range
      post  item = itemAt(pos) and selected(item) // inserted and selected
                  and count = count@pre + 1 // count increased from before
                  and movedUp (pos,count@pre) // items moved up
```

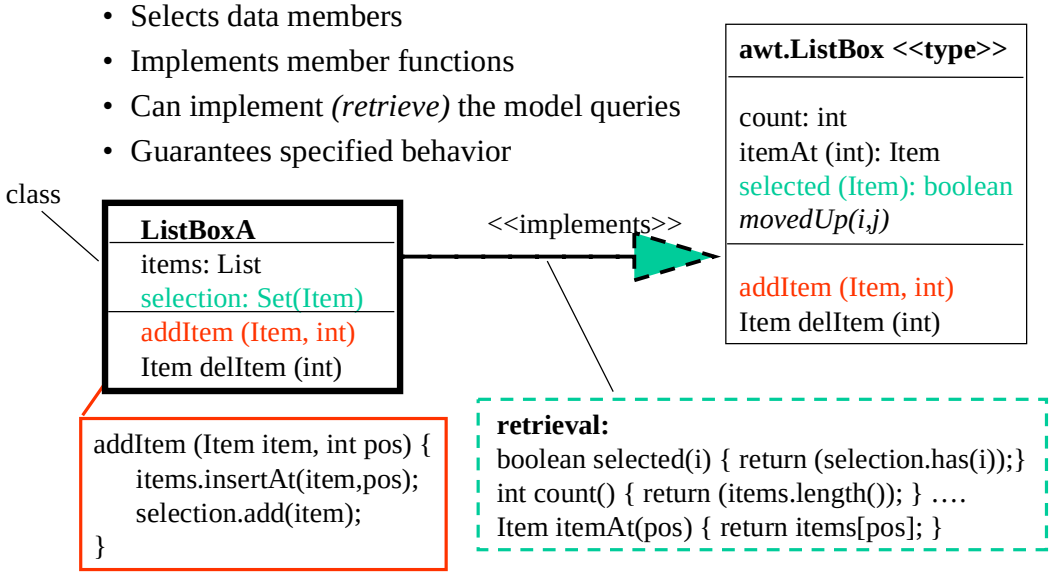

The operations can now be specified more precisely using the vocabulary provided by type-model attributes.

The precise syntax used here is not particularly relevant. With UML 1.1 and beyond, there is a standard syntax based on the *OCL* (Object Constraint Language) which is used to write such specifications.

The above pre/post would be described in OCL as follows:

Valid implementations of a Type

- Any valid implementation class
  - Selects data members
  - Implements member functions
  - Can implement *(retrieve)* the model queries
  - Guarantees specified behavior



Of course, there are many ways of implementing a type. Each implementation will choose appropriate data representations -- called instance variables or data members in most object-oriented programming languages -- and will then code the methods or member functions corresponding to each operation in terms of that representation. We have an implementation class written in terms of one set of instance variables, that claims to implement a type whose operations were specified in terms of another set of abstract attributes. We need a mapping from the implementation to the abstract specification, called a retrieval. In the diagram above, the client has, of course, understood and agreed to the type. The implementor programmed the class. Here is the dialog from the Design Review in which the class is being reviewed:

Client: Wait a minute, my specification was written in terms of count, items at different positions, and items being selected or not. How can I review your design if I don't even see them in your class?

Implementor: Well, there are actually all there. I have chosen to represent them differently so I could re-use the class List, and also get the right space-performance we need.

Client: Where is count?

Implementor: Count corresponds to the length of my "items" list.

Client: So do I now need to look at the implementation of List?

Implementor: No. In fact, I don't know the implementation myself, as I picked it from the library. However, it does have a specification, and count is defined there consistently with what we need.

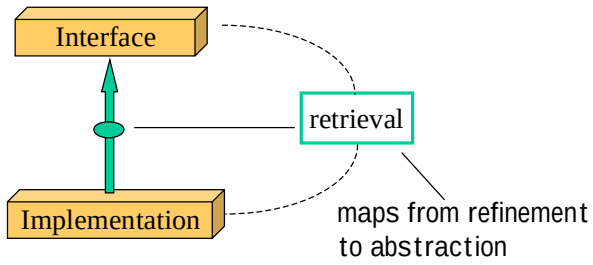
Note: the implementation itself relies on other types i.e. using their type models and specs

Client: What about selected(item) and itemAt?

Implementor: item is selected means the same as that item is in the selection set in my design

….And so on. A design review must justify these mappings, and the reasons for design choices.

Implementation: Conformance and Retrieval



- A class implements an interface
  - The interface defines its own model and behavior spec
  - The class selects its own data and code implementation
- The class is a *refinement* of the type I.e. *conforms* to it
  - It claims to meet the behavior guarantees of the type for any client
  - A *retrieval* (informal or formal) can support the claim
  - Implemented abstract queries (formal) can be used for **testing**

The distinction between class and type is one example of the Catalysis concept of *refinement* I.e. the same thing being described at two levels of abstraction, with a mapping from the concrete description to the more abstract one. In the case of class vs. type, this mapping includes a mapping from the concrete stored representation chosen to the abstract type attributes, and a corresponding justification (informal or formal) for the method bodies in the class implementing the specified operation post-conditions on the type.

The Power of Conformance/Refinement

- The notion of conformance and retrieval is very useful
  - It permits flexible *mapping* between from refinement to abstraction
  - It solves a very real problem:
    - *"I have just made some change to my code. Do I have to update my design models? Do I have to update my analysis models?"*
- Pick abstract model to conveniently express client spec
  - Implementation model must have correct *mapping* to the abstract
  - Encapsulates implementation without hiding specified behavior
- Even more powerful with *temporal refinement*
  - The abstract level describes an abstract action
  - The concrete level details an interaction sequence
  - The retrieval establishes the mapping between the two

28

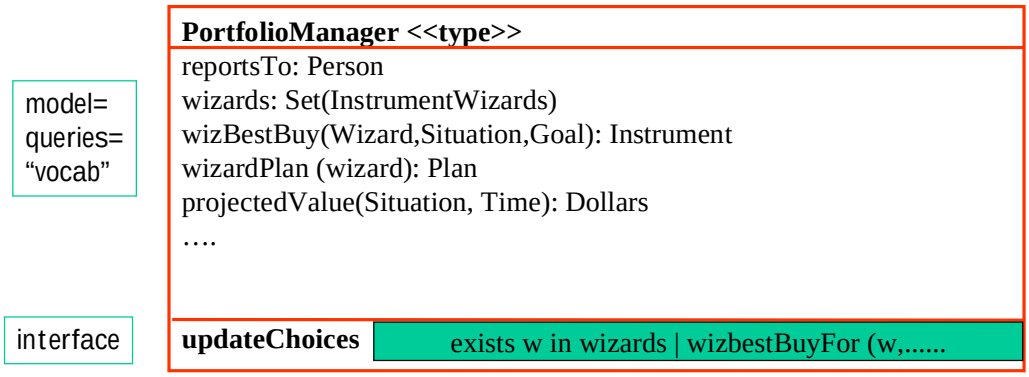
Refinement and retrievals solve a very real software engineering problem, specially in the face of today's trends towards iterative and incremental development cycles. Just because some system is being developed iteratively does not decrease the value of abstract descriptions of that system e.g. design or requirements models.

Many OO developers like to draw pictures of their code: boxes for classes, attributes for data members, operations for member functions, lines for inheritance or pointers between objects. While useful in a limited way, there is a serious risk that all such diagrams will be interpreted as visual representation of code. As a result of this view, an "analysis" model will be interpreted as a drawing of the code. When detailed decisions are made (or changed) in the code, it is not clear which of the models need to be updated. Without a precise criteria for propagating changes across levels of abstraction we have a maintenance problem.

As a result, either: (a) the cost of maintaining the models becomes too high, and the models quickly become obsolete and "die", or (b) the analysis models are reverse engineered from the code and lose their value as problem-centric abstractions.

The value of refinements and retrievals will greatly increase as our repertoire of refinements increases. In particular, *Catalysis* defines a basic set of refinements which include temporal and dialog refinements, model refinements, signature and "connector" refinement, which cover several useful forms of abstractions used in practice.

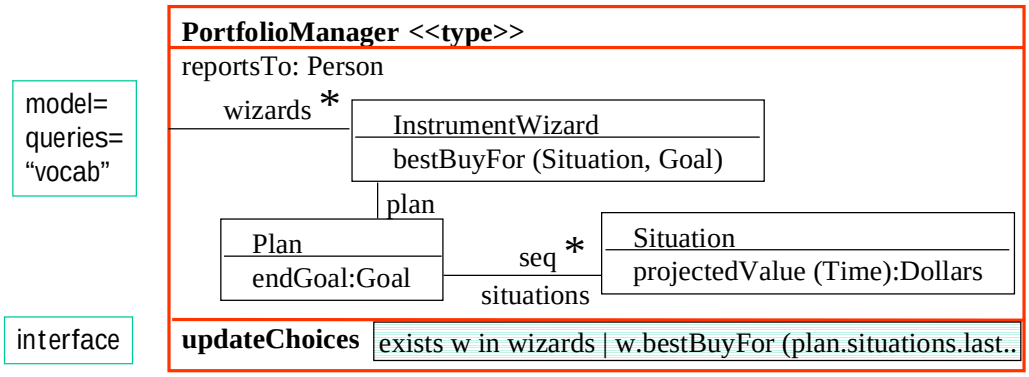
Complex Model Queries



- Complex behaviors use complex queries
  - Sets, sequences, etc. of other types
  - Related queries on other model types

In the case of complex type models, we do not want to list long lists of complex and parameterized queries. Instead, we will use a diagram form to express to same information.

Complex models will be shown visually



- Model queries may be depicted visually as *associations*
  - aPlan.situations: a sequence of situations
  - plan.situations.projectedValue(11:00): a sequence of dollars
- The spec for **updateChoices** can hence utilize:
  - reportsTo, plan, plan.situations, plan.situations.projectedValue(11:00),…

And here is what a type-model diagram looks like. It is important to read these diagrams as *abstract* descriptions of attributes of a PortfolioManager, defining *terms* that must be understood to understand the specification of operations of that object. None of the constructs in the middle section of that box represent data stored, or methods that must be implemented in any client-visible form.

However, as before, every correct implementation of this type must have some mapping to the concepts of *wizards*, *bestBuyFor*, *plan*, *endGoal*, *situations*, *projectedValue*, etc.

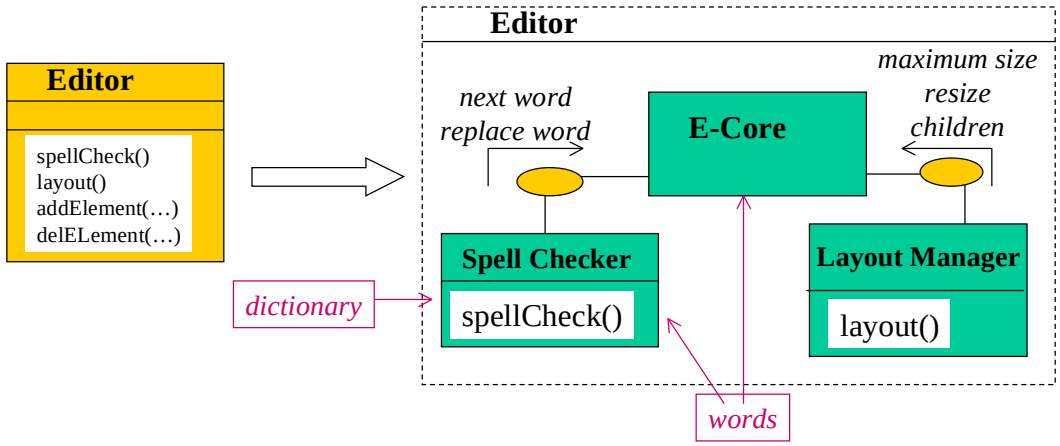
Outline

- Method Overview
- Component Specification - Types
- **Component Design - Collaborations**
- Component Architectures
- Refinements
- Business Example and Development Process
- Frameworks

We now know what it takes to specify an object or component externally as a *Type.*

The next section will show how such a type can be implemented internally as a *Collaboration* of several other components.

Component-Based Design



- Large component is a **type** for its external clients
- Implement it as **collaboration** of other components
- Specify these other components as **types**
- The child component models must map to the original model

We must first partition the Editor into constituent parts that will be implemented separately, but must collaborate with each other.

We choose to separate the spell-checking functionality, layout algorithms, and the core of the editor that maintains its structure and provides access to it to the user, and to the spell-checker and layout-manager.

The spell-checker and layout-manager expect very different services from the core. All the spell-checker does is request *nextWord* repeatedly, asking to *replaceWord* for any word it does not accept. The layout manager, on the other hand, perceives the core as a hierarchical structure of graphical elements (including text), which it can "modify" at certain points for purposes of layout e.g. to break a line or page, and to re-locate a floating table. The operations it uses are quite different, and related to sizing and nested structure of tables, lines, words, etc.

Note some points about this design:

The terms in our original type model of an editor are now split across the designed components. The spell-checker is the only one that has any concept of a *dictionary*. Both the spell-checker and the core have to understand and exchange *words*. The layout manager probably also manipulates words, but really only treats them as elements that contain sequences of child elements (characters), and that can be broken at certain places for layout reasons.

Also, the core offers two interfaces to its collaborating components. Thus, we could well characterize this core as *two different types*, one as seen by the spell-checker, and the other as seen by the layout manager. This re-inforces a point we made much earlier; an object or component plays different roles with respect to different collaborators, and exhibits different types to each.

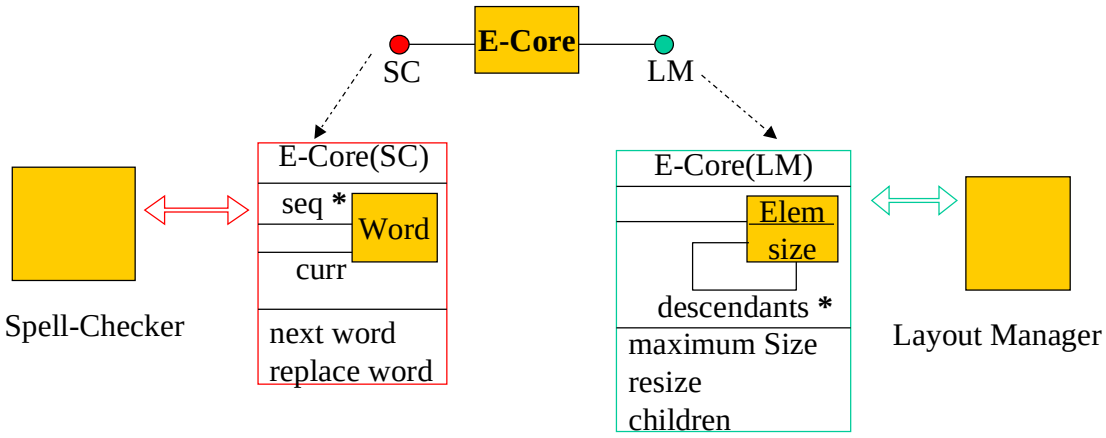
Reasoning for Component Partition

- In this particular example of Editor
  - Spelling and Layout are quite *orthogonal* to each other
  - Layout can be *separated* from the basic structure manipulation
  - If we build a Spell Checker we can *use it in lots of applications*
  - We can *buy* many spell checkers
  - We can imagine many *variations in layout policies*
  - Both are examples of the *strategy* design pattern
- *Design the **interface** to make the separation effective*
  - Spell Checker sees “a sequence of replaceable words”
  - Layout Manager sees “a nested structure of things to position and size”

Here are some possible reasons for the partitioning.

Our focus in this bit is not on the heuristics and guidelines for good partitioning, so we will keep this short. The essential reasons in this case have to do with re-use, flexibilty, and buy-vs-build

Each Component implements many Types



- Components offer different interfaces to each other
- Each interface has a different supporting model
- The implementation **refines** each interface spec

If we try to specify just the editor core, it exhibits two different types.

E-Core(SC): the editor core as seen by the spell-checker. The only two operations here are *nextWord* and *replaceWord*. If we specify these two operations, we may end up with something like this:

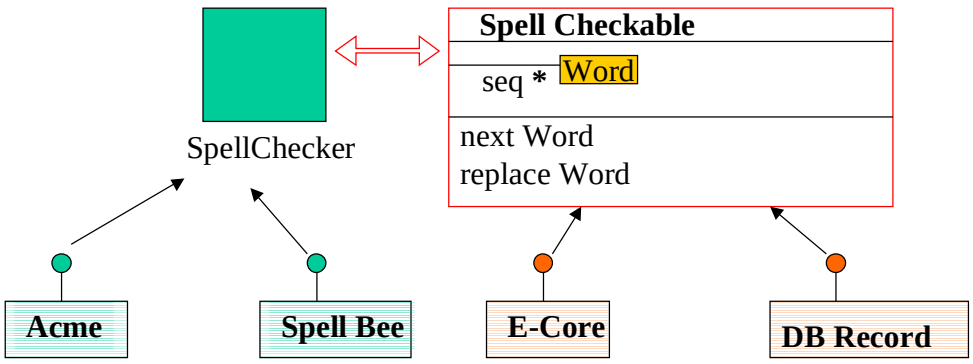
> *nextWord*: the current word position is moved forward by one word, and the word at that position is returned to the spell checker. If there are no more words, null is returned.
>
> *replaceWord(replacement)*: the word at the current word position is replaced by the replacement.
>
> If we try to make these two specs a bit more precise, we find that our vocabulary must define the concept of *current word position*, and some concept of *the sequence of words*. Hence, our type model of the editor core looks as shown in the figure on left.

E-Core(LM): the editor core as seen by the layout-manager. The operations here are *maximumSize* and *resize* and *children*. If we specify these operations, we may end up with something like this:

> *maximumSize (element)*: the size of the element in question.
>
> *resize (element, size)*: sets the size of the element.
>
> *children (element)*: returns the set of children of the element.
>
> If we try to make these specs a bit more precise, we find that our vocabulary must define the concept of *size of element*, and some concept of *the children of an element*. Hence, our type model of the editor core looks as shown in the figure on right.

Of course, anyone who implements an E-Core must implement *both* these types. Whatever representation it chooses, it must be able to *behave as though it was comprised of a sequence of words* (for the spell checker), and *behave as though it consisted of a nested tree of sized elements* (for its layout interface).

Type-Based Components are “Pluggable”



- Any component that provides the correct interface (operations and apparent type model) can plug-into another component

The entire reason for designing in terms of collaborating components with multiple interfaces is to provide more flexible and de-coupled designs.

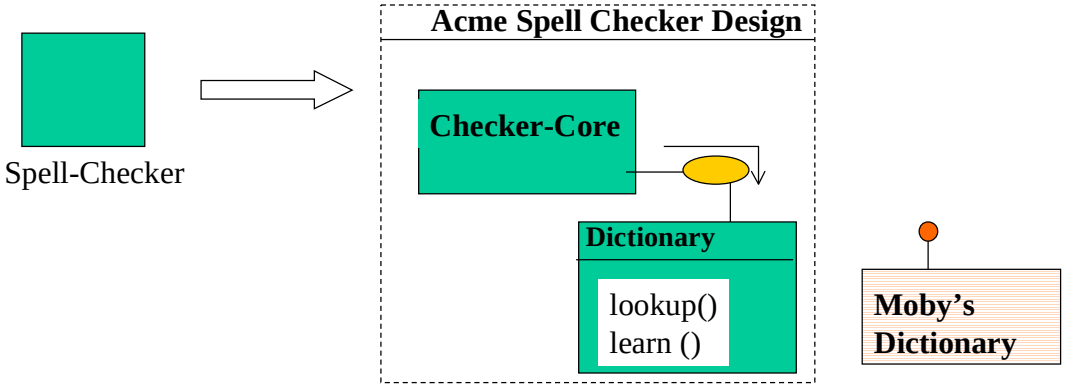
If the editor-core implements our *SpellCheckable* interface, it can plug into an correct implementation of the *SpellChecker* type. This is because the *SpellChecker* type was specified to work with any object that implemented the *SpellCheckable* interface. We should be able to plug in or out either *Acme* or *SpellBee* spell checkers.

More interestingly, are there other things besides an E-Core that we could spell check with any spell checker? Could we spell-check a spreadsheet? A database record? An email message?

Yes! The only requirement is that any of these objects must implement the *SpellCheckable* interface i.e. they must be capable of *masquerading as though they consisted of a sequence of words.*

This approach buys a tremendous amount of fexibility in the software architecture, if the interfaces are chosen judiciously.

Implement a Component using Others



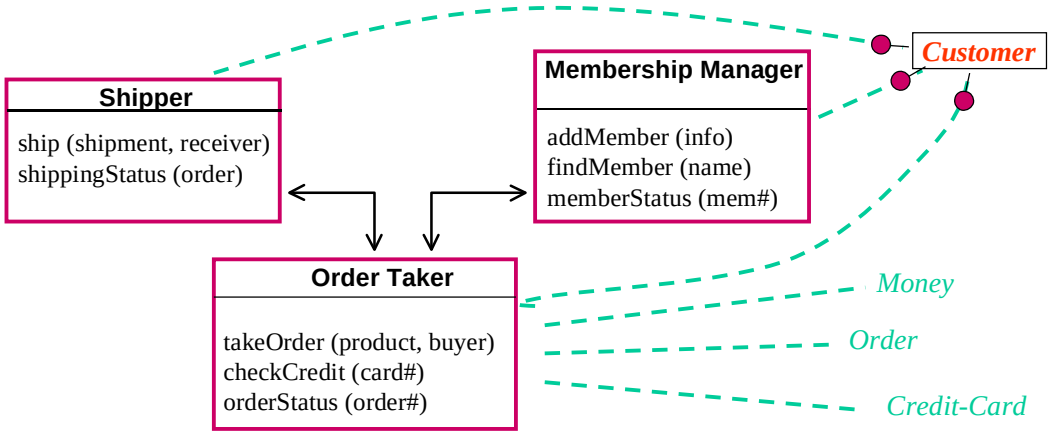
- And we can now plug-in and out different dictionaries, languages, ….

And this component-based design continues recursively.

Thus, a *SpellChecker* may be built so it can use any implementation of a *Dictionary*.

Large-Grain Business Components

- "Executable" component = large-grained object
- Components configurations tend to be more static
- One component may be built from several classes
- Underlying objects implement several types e.g. ***Customer***



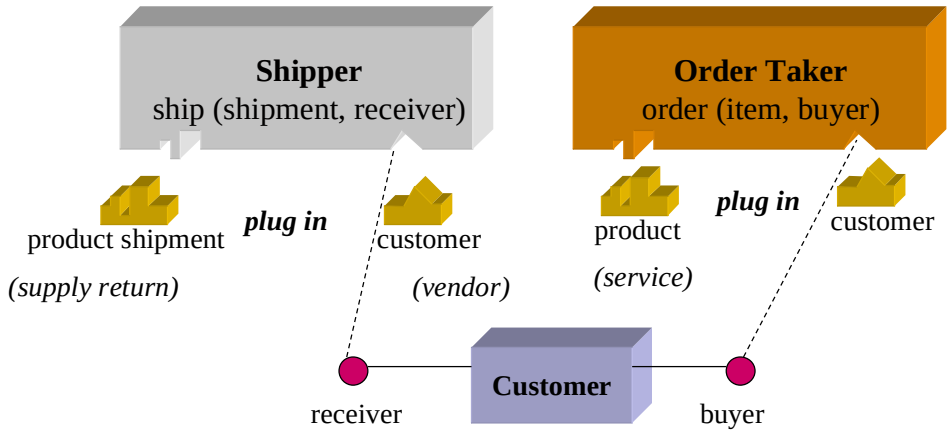
A "component" can span many levels of granularity. In fact, even our legacy mainframe applications can be considered components -- they were just not very easy to compose with other components, as the ugly solutions of screen-scrapers and terminal emulators demonstrate.

In particular, medium to large-scale components can constitute entire business functions. These components tend to be somewhat more static in their configuration and interconnection with other components, and each such component will be built from the equivalent of several OOP "classes".

These components will themselves implement multiple interfaces. Moreover, they will utilize underlying shared objects, such as *Customer*, and will require different interfaces from that object itself.

“Frameworks” as Components

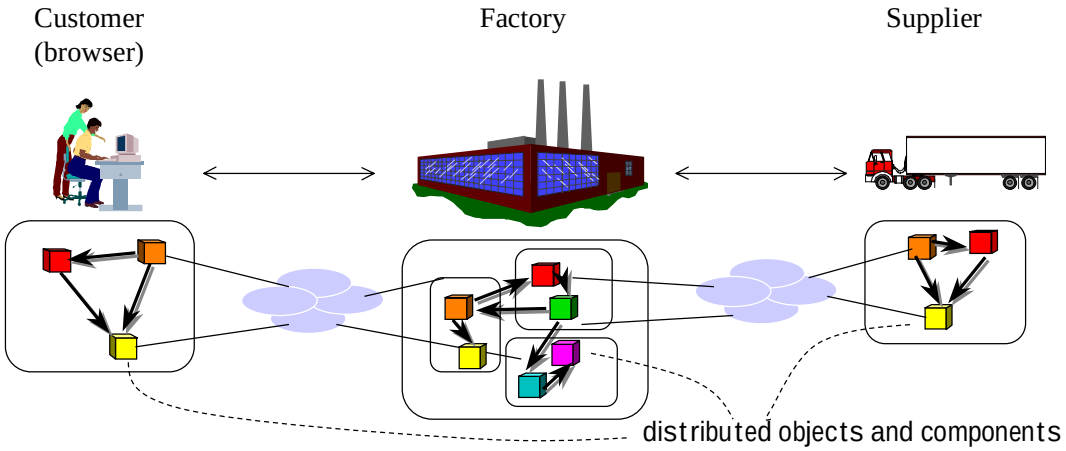
- A large-grain component designed with “plug-points”
- Application will “plug” domain objects into plug-points
- “Plug-in” based on interface, sub-class, delegation, etc.



The most flexible components are *“frameworks”* i.e. components that provide the skeleton of some business or application function, but do so in a manner which permits “plugging-in” of customization pieces to adapt the component to a specific use.

The illustration shows how a *Shipper* component may be adapted to ship any *Shipment* to any *Receiver*. It can be customized to ship product shipments to customers, or to ship returned supplies to vendors. Seemingly trivial, this example is actually very typical of the unnecessary dependencies built into many legacy systems, in which it might be an impossible task to use the same shipping component to handle both customers and vendors.

Federated Components - The Virtual Enterprise



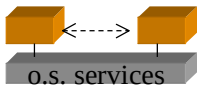
- Supply-chain and other cross-business federation enabled
  - Build on internet or similar technologies
- Requires open distributed objects and components
  - Demands supporting infrastructure standards and component re-use

The scale and impact of collaborating components is most dramatically illustrated by internet technology. Not only can components be federated across different business functions within the same company, but they can can be extended (via appropriate security mechanisms) to components in the customer's and supplier's software systems as well. The *"virtual enterprise"* is rapidly becoming a reality, based on underlying technologies of collaborating components and distributed objects, with the appropriate infrastructure standards.

Evolution of Component Interactions

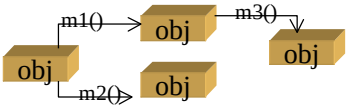
- Mainframe (host) applications
  - quite unnatural, not designed for this

- O.S., databases, inter-application communication
  - essential support of “system services”

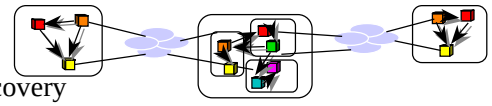


- Unix pipes
  - effective but limited “connection” model

- C++ objects (or Smalltalk, ...)
  - effective, limited scope, early OO mistakes



- COM, Corba, JavaBeans, …
  - tackles broader scope and issues
  - interface-centric, distribution, services, discovery



Just as a bit of perspective, consider how “components” have evolved over the past 40 years. It all boils down to collaborations between components -- the essence of open and extensible systems.

In the beginning there was the mainframe. And we wrote monolithic host applications for the beast.

Unfortunately, these host applications did not make very good “component” citizens. Specifically, they were not amenable to integration with other such components, for two reasons:

> The component granularity itself was way too large. Re-use at this level of granularity is a long shot!
>
> The components were written very specifically for input and output from a human via a dumb terminal.

As a result, we had to write fairly ugly pieces of code, with names like “screen-scrapers” and “terminal emulators”, to fool the software components into thinking they were interacting with such a dumb terminal, while in fact intercepting the traffic to route to another component.

Then came operating systems with services for inter-application communication, enabling the separating of networking services, and of database servers from applications, and the entire client-server world was born.

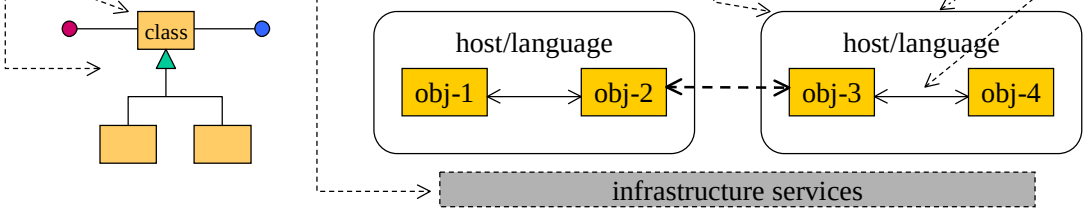
In the Unix world, a particular component architecture was taking shape: the *pipes and filters* model. In this model, each component (typically an executable unit) consumed and produced a stream of bytes or characters. Composing components was a relatively simple matter of connecting their input and output streams together, with appropriate “infrastructure” components for providing splits and joins of these streams. Elegant, but very limited.

Along came object-oriented development. The basic component unit here was an object, and component interaction took place by explicit inter-object messaging. The model was fairly general, but was limited due to several “mistakes” of early OO technology: language specific focus, lack of standard infrastructure services, lack of attention to distribution, etc.

Today’s component technology takes us a step further, filling in many of the missing pieces in traditional OO technology, while broadening the scope beyond language and distribution boundaries with infrastructure services.

Are Components and Objects Different?

- Most components can be described as objects
  - both emphasize encapsulation, interfaces, polymorphic late-bound connections
  - most new components will be built from traditional objects inside
- So, what did traditional object-oriented development do wrong?
  - undue emphasis on class and implementation inheritance rather than interfaces
  - just one basic form of "connectors" -- explicit message invocations
  - unfortunate language-imposed dependencies on class source code -- *.h* files
  - boundaries such as host, programming language visible to components
  - infrastructure services ignored -- persistence, transactions, security, discovery…
  - component can be larger than a traditional single "object" or "class"



Are components and objects different?

In a single word: *NO!*

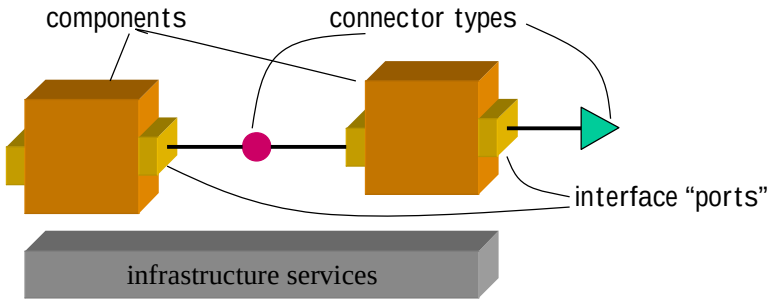
All of today's component technology can be described in terms of underlying interacting objects.

However, components have raised the bar for software construction in some important ways:

- they have changed the focus from classes and implementation inheritance to interfaces and composition
- broader model of "connecting" components from the traditional explicit method call of OOP, to more general models including events and properties today, and even more general connectors in the future
- component technology typically crosses languages and distribution boundaries
- component technology pays more attention to standardization of infrastructure services that are common to all applications
- they focus on somewhat larger-grained units of re-use than a traditional "object"

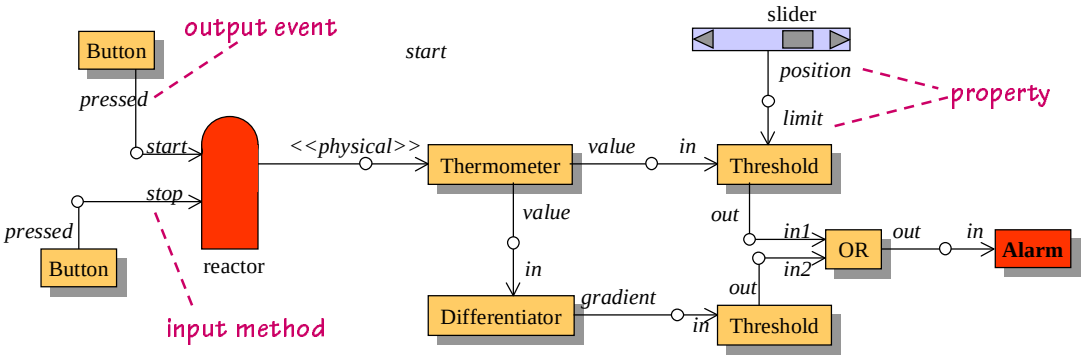
Component Terminology

- *Component:* Software chunk that can be "plugged" together with others
- *Connector:* A coupling of a particular nature between *ports* of two *components*
- *Port*: The "plugs" and "sockets" of an individual component
- *Component Architecture:* (a) Standard port "*connector*" types and rules
- *Component Infrastructure:* (b) Standard services for components and connectors
- *Component Kit:* Components designed to work together with common architecture
- *Packaging:* Packaging of a component with associated specs, resources, tests, docs, ...



Here is some terminology from component technology. This terminology is somewhat young, and is still in evolution. Our definitions may be somewhat broader than that typically used in the popular press.

Components generalize “Connectors”



COM, Java Beans

- *Events* out of a component, or *in* to a component (triggering a *method*)
- Input *properties* that are kept in sync with output properties
- *Workflow* “transfers” where an object is transferred
- *Replication* -- where information is copied and kept in sync

- Standard *objects* and *method invocation* underlie all of the above

This example illustrates how component technology provides for more general forms of “connectors” between components than simple explicit method invocations.

The figures shows examples of *output events* in one component being linked to *input methods* to invoke on another component, as well as examples of *properties* linked across components.

Why Components?

- A economically and technologically practical way to organize and package an object-oriented (or other) system
- Developed, marketed, and maintained on a component basis
- Support capabilities that are impractical for “small” objects
  - Interfaces accessed through different programming languages
  - Components interact transparently across networks
  - More cost-effective to maintain since they do more than “small” objects, and less than “monolithic” programs
- Each component could itself be a candidate “product”
- Component partitioning enables parallel development

One may well ask: what is the fundamental difference between an object and a component?

The basic answer is *None*.

However, historically, objects have focused on purely business-domain concepts and their interactions with each other via message passing, and have not identified larger-grained components like *Shipper* and *OrderTaker*. Object technology also initially went overboard in using *class inheritance* as a mechanism for code-reuse, sometimes at the cost of identifying narrow interfaces for collaborations between pieces, and at the cost of unnecessary coupling between superclass and subclass.

On the other hand, component enthusiasts emphasize interfaces more strongly. Components can often be larger-grained than *traditional* objects, although most of these components can themselves be considered to be (singleton) objects. Thus, traditional OO could well have numerous instances of object types like *customer*, *product*, and *order* interact to provide all the functionality of order-taking, membership management, and shipment, without using objects like the singleton *order taker* or *shipper*.

The nature of the *connector* between *components* is frequently richer than for traditional objects. In most OOP languages, objects interact exclusively by messages -- an unfortunate term, which actually means a synchronous invocation of a method on the target object. However, it it frequently useful to consider richer kinds of *connectors* e.g. a *property* of one object may be directly connected to a *property* of another, always keeping them in sync. Alternately, an *event* in one object may be connected to a *method* in another, causing that method to be invoked each time that event occurs. While all these are likely implemented with method invocations at the lowest level, they can modeled more effectively as a different kind of architectural *connector* than straight method calls.

Those with strong experience in building host-based monolithic systems and transitioning to components, frequently miss the *collaborative* nature of components, and the fact that even large-grained singleton objects need, at the lowest level, finer grain objects like customer and product.

Outline

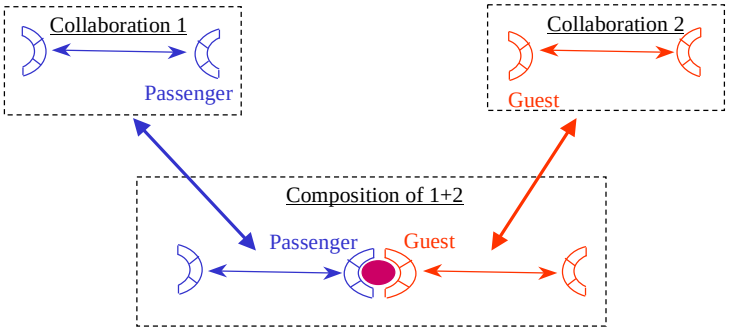
- Method Overview
- Component Specification - Types
- Component Design - Collaborations
- **Component Architectures**
- Refinements
- Business Example and Development Process
- Frameworks

45

We have seen how a component can be externally specified as a *Type*, and how its internal design is a *Collaboration* of other parts.

This section shows how the basic idea of an object of component having multiple interfaces leads to a fundamentally different approach to architecture, in which the architectural elements are actually patterns of collaborations that realize specific services.

Interfaces lead to *Collaboration Design*



- An object offers a different interface in each role it plays
  - Related objects talk through related interfaces
- Interface-centric design makes us focus on **collaborations**
  - A collaboration defines related roles and interactions
  - Provides excellent basis for composing patterns / architectures

First for a quick review…

We start with the "objects 101" version of an object, the kind you see in a David Taylor book for managers. It looks like a little jelly donut: *"I am a well behaved object, this here inside is my data and no one else can see it; this surrounding it is my services that others can access"*

The first thing we recognized was that an object offers different interfaces to different collaborators. We sliced the outside of the jelly donut into smaller sectors, as shown above. The object in the center could be a *Person* with an interface for being a *Passenger* on an airline, and one for being a *Guest* in a hotel.

The next question is: *who would be interested in the Guest interface of a Person?* Probably not the Airline. However, the hotel is interested. In fact, the hotel itself also plays other roles, such as an *Employer* role with respect to its staff, and a *Taxpayer* role with respect to the tax authorities. These other "faces" of a hotel have no interest in the guest role of a person. Hence, it is the *BoardAndLodge* of the hotel that works together with the corresponding *Guest* interface of a person.

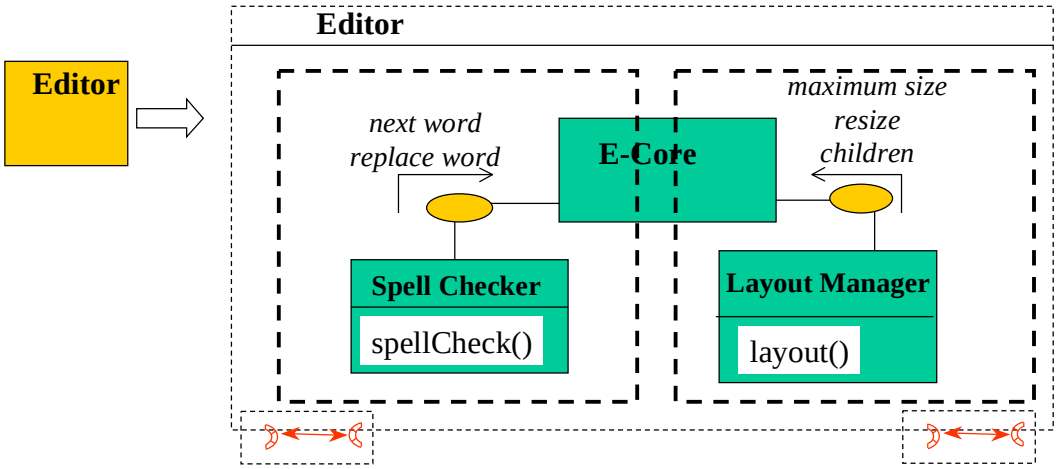
Similarly for the *Passenger* interface of a person, and the *TravelProvider* interface of an airline.

Hence, these interfaces and types belong in groups. The most meaningful design units consist of *Guest+BoardAndLodge*, rather than just *Guest* in isolation i.e. our design elements are packages of interfaces!

On a more subtle and advanced note, a single object could provide both the *TravelProvider* and the *BoardAndLodge* interfaces for a single person -- consider a luxury ocean liner that ferried passengers before the advent of airplanes. It would still be true that the *TravelProvider* portion would care about the *Passenger* interface, while the *BoardAndLodge* portion would care about the *Guest* interface.

(no offence implied; Dr. Taylor's 2nd edition makes very good reading, and the 1st was a good overview)

Recap - Our Collaboration Design



- Large component implemented as **collaboration** of components
- The design can itself be factored into 2 partial interactions between roles

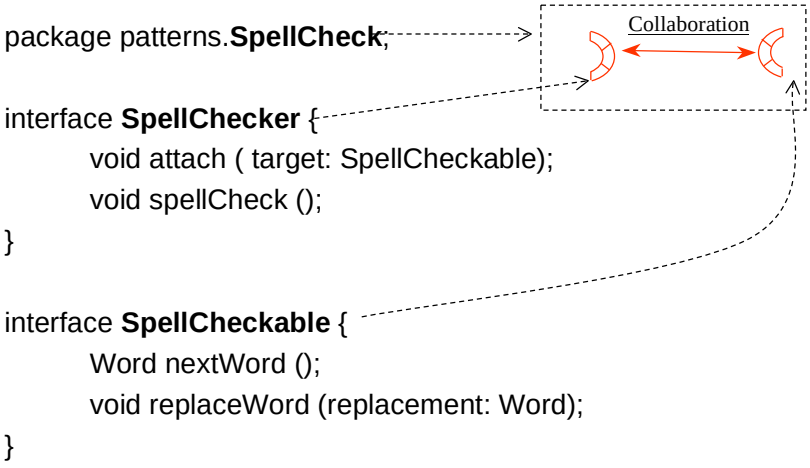
Well, what kinds of *architectural elements* have we used in our design of the Editor?

To refresh your memory, here is the partition we came up with: spell checker, editor core, and layout manager.

There are two distinct collaborations here, one to provide the spell-checking service, and the other for layout management.

Collaborations as Architectural Units

- A collaboration is an architectural package that contains:
  - a set of related interacting types
  - characterized by set of interfaces with behavior specifications



A collaboration is an architectural unit or package that contains a set of related interacting types, characterized by the appropriate types (interfaces with behavior specifications).

This slide shows some Java code for the spell-checking collaboration (well, pseudo-Java).

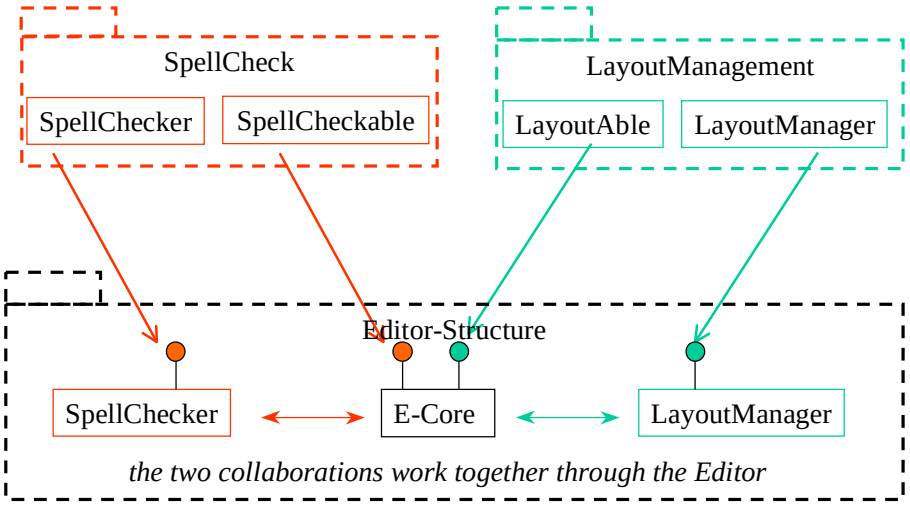
The package,called *patterns.SpellCheck* simply to emphasize that this is a design pattern for implementing spell checking in any application (and to set the stage for an upcoming discussion), contains two interfaces:

SpellChecker: any object playing this role must be able to *attach* to a target that provides the *SpellCheckable* interface, and then to *spellCheck* that target.

SpellCheckable: any object playing this role must be able to provide sequential access to the words it contains, and to replace the word at the current position in that sequence.

Note that this package is *purely a design description*. It contains no implementation code at all. If we wanted, we could provide a default implementation class for each of these interfaces. We would most likely put this default implementation into a separate package. A particular implementation could decide whether or not to use that default implementation; if it did not, it could still implement this design pattern.

Composing Collaborations



- Defines effective ways of *using and reverse-engineering* patterns

The spell-check design pattern simply describes one design for one service. But, of course, no application consists simply of spell-check. An editor would provide operations for *open, edit, save*, in addition to spell check. An email application would provide *reply, send, delete, …* in addition to *spell check.*

Thus, each application utilizes several design patterns, composed and mutually interacting in ways unique to that application.

In our editor, we utilized two distinct patterns: one for spell check, the other for layout management. Each of these is an architectural element, described in its own package.

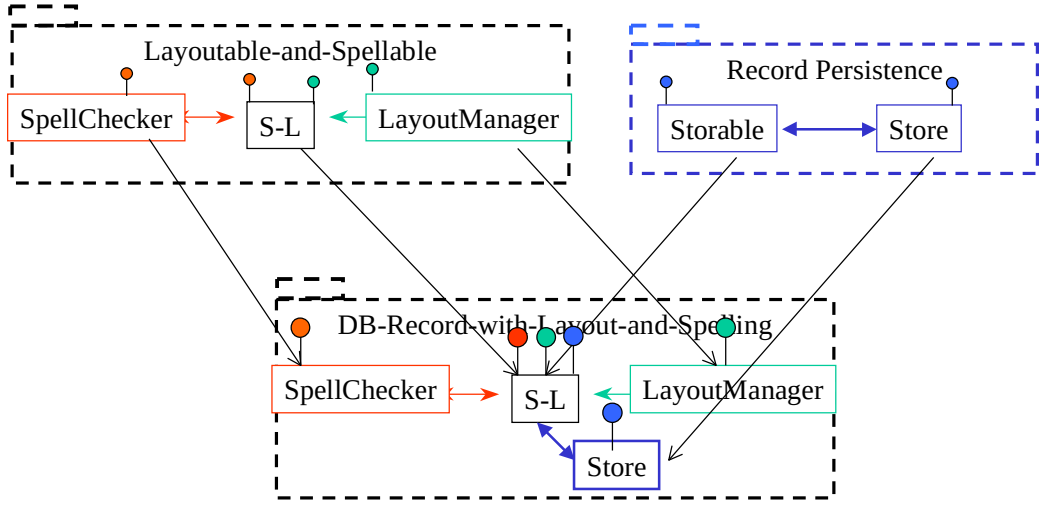
The Editor package could be defined as a composition of these two patterns. Specifically, the editor core plays a role in each of these patterns, implementing one interface for each.

Although each pattern was described independently of the other in separate packages, the collaborations are no longer independent when composed in the editor application. For example, when spell checking a document, we may replace a short word with a longer one. This may trigger some changes in layout, and cause re-computation by the layout manager.

All this means that when we describe and compose patterns, we must also provide more “open” ways to hook them into other patterns when composed, so that effects in one can have effects on other as-yet-unknown collaborators.

The most interesting result of this slide is that the architecture of a component-based design is captured by the *collaborations* that have been applied or composed in it. Documenting a design means documenting the patterns of collaborations that have been used in it.

Composing Collaborations - Recursively



- Leads us to *interface-rich designs* and *design composition*

The composition of patterns continues recursively.

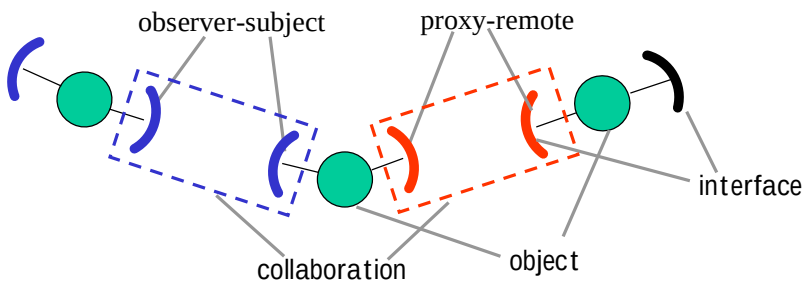
Suppose we wanted to build a data-base system that supported persistent database records (naturally!), but also allowed for spell-checking and automatic layout of the contents of a records e.g. smart line-breaks and indents of the fields.

We could re-use the combined pattern of spell-checking and layout-management from our editor -- except, of course, we would probably use its *design*, as opposed to the *E-Core* class that implemented editor-specific structure and operations.

We could combine this with a bare-bones *Persistence* pattern, which required two types for its two roles: *Storable* (the things that must be stored), and *Store* (the medium, file, database, …) that it will be stored in. The composition of these patterns is shown in the slide above.

This example used very service-specific patterns. As we will see, service-independent patterns emerge when we abstract away certain application specifics.

Collaborations as Design Patterns



- An object offers a different interface in each role it plays
  - Related objects talk through related interfaces
- Generic collaborations often represent **design patterns**
  - Can be defined as sets of interfaces in a Java package
  - In Catalysis, augment with behavior model

The now-classic work of design patterns in the *Gang-of-Four* book by E. Gamma et al, consists largely of collaborations that effectively solve design problems, abstracting away from the domain-specific characteristics.

The slide above shows how two common patterns -- *Subject-Observer* and *Proxy-Remote* -- could actually be composed together in an application. The object in the middle is playing two roles:

subject: with respect to some other object acting as its *observer*

proxy: with respect to another object acting as the *remote*

These patterns can also be characterized as collaborations in Catalysis. Some of them, however, will need the additional expressive machinery of Catalysis *frameworks* to abstract away the domain specifics without a loss of precision.

Code-level Collaborations: Java Packages

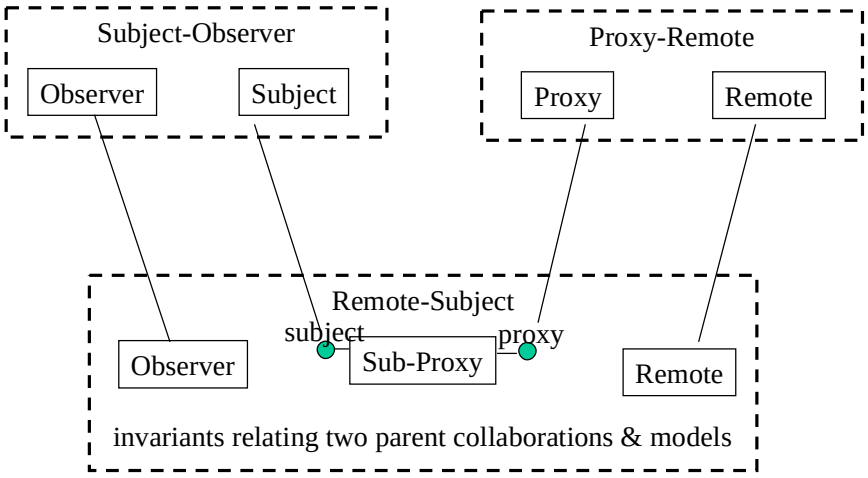
- A collaboration is:
  - a set of related interacting types
  - characterized by a set of interfaces with behavior specifications

```
package patterns.Observation;

interface Observer {
        void update (Subject s, Object changeInfo);
}

interface Subject {
        void register (Observer o);
        void deRegister (Observer o);
}
```

The *Subject-Observer* design pattern would appear in Java in a single package, much like our earlier *SpellCheck* service pattern appeared in a package. It contains the interface definitions for the two roles.

In Catalysis, these could additionally have behavior specifications attached to them, to capture the required behavior and interactions between the objects playing these roles e.g. when does a subject invoke the *update* method on its observers?

Composing Collaborations



- Defines effective ways of using and reverse-engineering patterns
- Can use Java *packages* for modularizing collaborations

And, of course, these kind of design patterns can also be composed.

This particular composition forms the basis for (some part of) 3-tier architectures. The UI elements act as the observers of the domain objects, which are their subjects. These domain object, in turn, are actually proxies for the data itself, which is represented by some objects across the network resides on the 3rd tier in some database.

Class Instances play Roles in Collabs

- Instances of a class play multiple roles
- The roles are characterized by interfaces in collaborations
- Java packages can exploit this (but lack “generics”)

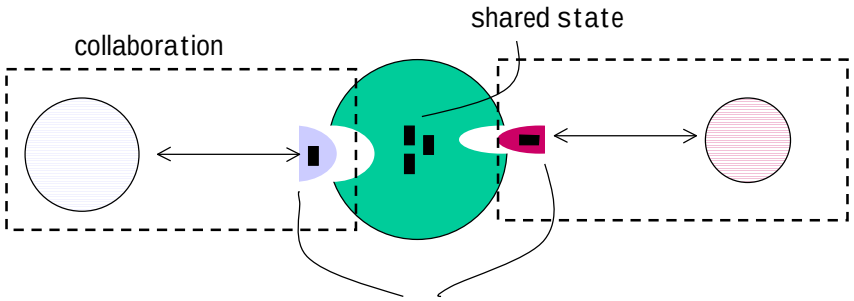
```
package myApplication;
import patterns.Observation.*;      // patterns.Observation(Order)
import patterns.Distribution.*;     // patterns.Distribution(OrderData)

class Order implements Subject, Proxy { ……. }

class OrderWindow implements Observer { …… }

class OrderData implements Remote { …… }
```


Here is some Java code for objects that implement a 3-tier architecture in a very explicit manner, importing the architectural patterns involved, and implementing the interfaces required by those patterns.

The class **Order** forms the middle-tier. The **OrderWindow** forms the client-tier or UI. And the **OrderData** forms the 3rd tier on the database server.

Code-Level Collaboration Composition

- We can carry role-structures into the implementation
  - A role-object for each role played
  - Design rules for dealing with (split) object-identity issues
  - Each role-object implements its part of its framework spec
  - A shared object for the “real” object
  - Uniform observation or notification mechanism between the two



Our examples so far illustrate the composition of design elements i.e. pure interface descriptions of collaboration patterns. They did not, so far, discuss the reuse of implementation code.

One way to reuse code is to:

Implement a class for each role in a collaboration. Design each role object so it expect to be “plugged into”, or connected to, a “real object” whose role it will represent.

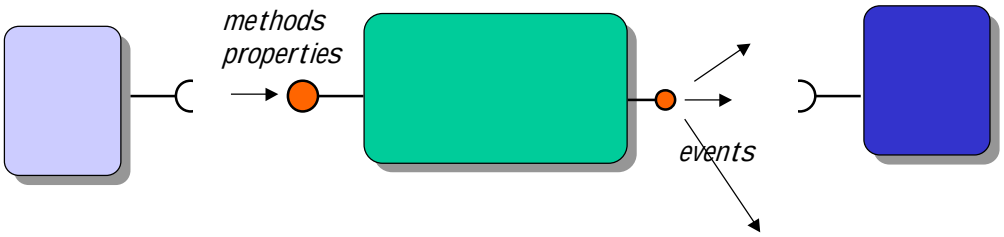
When these roles are “overlaid” on a single object in an application, due to composition of collaborations, represent that “single” by a group of objects:

One “central” object that represents the domain abstraction of interest

One “role” object for each role that domain object is supposed to play in each collaboration. This role object is an instance of the class that implements just that one role in its collaboration.

All the role objects share the same domain object. Interactions between role objects take place mostly through the shared state of this domain object.

Design Styles for Code Components



- Components can be composed together
- Each component should define
  - Abstract attributes which can be read and set
  - Services which can be requested from that component
  - Services required from other connected components
  - Changes ("events") that component is capable of notifying
- The last two are problems
  - Traditional object descriptions only focus on services provided

To re-use implementation code when programming with components, we need to be able to compose that code when we compose collaborations.

Of course, all the normal machinery of classes and inheritance is available when designing with patterns. However, there is one problem. The clases for one collaboration are design independently of the classes for another. When the collaborations are composed, however, and we end up with a single object playing a role in more than one collaboration, there must invariably be some interaction between the collaborations.

How will the interaction take place, if all the framework classes for each collaboration were defined without knowledge of other collaborations?

This problem actually highlights one of the shortcomings of traditional OO design. Its focus was mostly on the *incoming requests* that an object can service i.e. the messages on its interface that could be invoked by another. It did not explicitly consider, or document, its *outgoing* requests i.e. either the *services* it required from others in order to properly provide its function, or the *events* that it was capable of raising so that other interested objects could respond as they needed.

In order to facilitate better composition of collaboration implementations (and, incidentally, this also helps with composition of the specifications of each collaboration), component standards have moved towards a more explicit description of the *events* a component can raise. That way, when two components that each implement a role in two different collaborations are composed together and should act like they were one object, each one can raise events that the other can respond to.

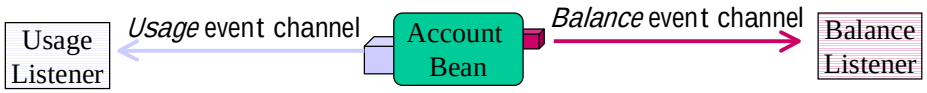
Catalysis provides very effective ways to specify events raised as part of a component type specification, and to compose components more conveniently using these events. In addition, it allows for much more general forms of *connectors* between components, in addition to today's emerging standards of *properties, methods, and events* e.g. connectors for workflow, transactions, replication, pipelines, etc.

Component “Events” e.g. Java Beans

- An event is an interesting change you can subscribe to
- Events grouped into *event channels* by *Listener* interfaces

```
interface BalanceListener extends java.util.EventListener {
        void overdrawn (BalanceEvent);
        void balanceOK (BalanceEvent);
}
```

- Bean may offer different *event channels*



- Standard convention for registering event listener:

```
void addBalanceListener (BalanceListener f);
void removeBalanceListener (BalanceListener f);
```


Java utilizes a particular style for documenting and using events in its *JavaBeans* standard for components. Although Java, the language, is still a normal OO language, focused on *incoming* requests rather than *outgoing*, this style effectively addresses the shortcoming.

Suppose we were building a component that represented a bank account -- we will call it our *AccountBean*. The basic steps are as follows.

Identify the interesting events you will raise: e.g. balance became overdrawn, balance became OK, account is seeing excessive activity, account has become dormant

Group these events into separate “channels”, based on different clients probably being interested in different subsets of these events:

overdrawn, balanceOk: an account-balance event channel

hyperActive, dormant: an account-activity event channel

Document each channel using an *EventListener* interface i.e. anyone who wanted to listen in on that channel had better expect those events, so had better implement that interface

interface BalanceListener { void overdrawn (BalanceEvent); void balanceOK (BalanceEvent); }

interface ActivityListener { void hyperActive (ActEvent); void dormant (ActEvent); }

Provide an *add/remove* pair of methods to the AccountBean for each channel, for listeners to “tune-in” and “tune-out”

addBalanceListener (BalanceListener); removeBalanceListener (BalanceListener)

addActivityListener (ActivityListener); removeActivityListener (ActivityListener)

Outline

- Method Overview
- Component Specification - Types
- Component Design - Collaborations
- Component Architectures
- **Refinements**
- Business Example and Development Process
- Frameworks

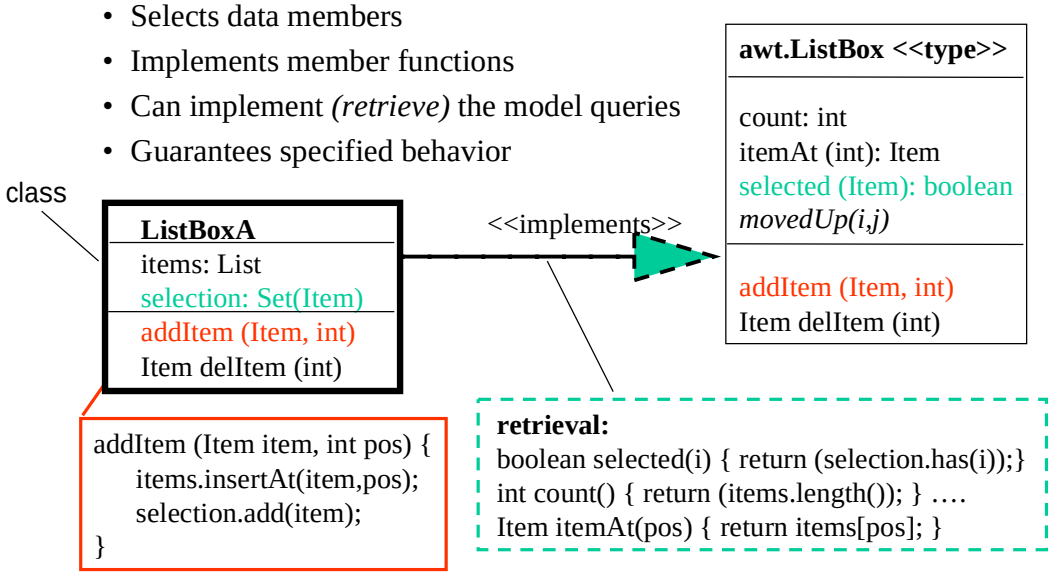
58

This section will introduce the concept of *Refinement* as used in Catalysis.

Refinement is a very central concept. Much of software development can be considered to be the production of descriptions that are refinements relative to things produced earlier (top-down development), or that are abstractions of things produced earlier (bottom-up development).

Class Vs. Type -- A Basic Refinement

- Any valid implementation class
  - Selects data members
  - Implements member functions
  - Can implement *(retrieve)* the model queries
  - Guarantees specified behavior



We have already seen how a **class** is a refinement of a **type**.

The type abstracts away many representation and algorithmic choices. It expressed operations abstractly in terms of their net effect (pre/post conditions) on abstract representation attributes. The class selects instance variables, and writes methods to manipulate those instance variables and implement the operation specs. A design review **must justify** the mappings from class representation to type attributes to defend the implementation Vs. the spec, and the reasons for design choices.

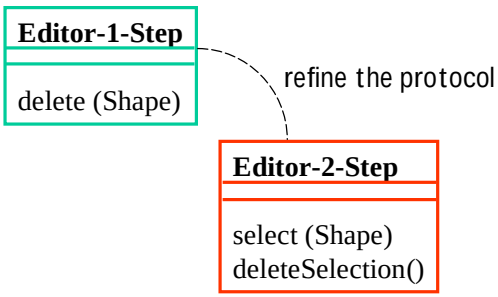
Beyond Subclass vs. Subtype

- Subtype vs. subclass: a much discussed distinction
  - Type defines a set of objects by their behaviors
  - Class defines interface and implementation of type(s)
  - Subtype defines a subset of a type by their behaviors
  - Subclass inherits interface and implementation
  - Both focus on behavior of **an** object
- ***But most interesting designs involve multiple objects***
  - Roles, relationships, and interactions define architecture
  - Many design decisions involve and affect multiple objects

Designs and architectures involve multiple objects and their interactions. No amount of discussion and sophistication with regards to classes, subclasses, types, and subtypes will help us directly in describing, composing, and refining architectural aspects.

Subtypes and Refinements

- Sub-types refine (and retain) guarantees made by super-types
  - The concrete implements, and can *retrieve* to, the abstract
  - Any sub-type implementation meets all super-type behavior guarantees
  - *Clients remain blissfully **unaware** of any change*
- Here is a common refinement: is it a sub-type?



Consider the two descriptions above:

Editor-1-Step: at an early stage in requirements for an editor, we decide that it must support deletion of a shape. We document a type with the appropriate operations. We might even formalize this operation with a spec, and a supporting type model.

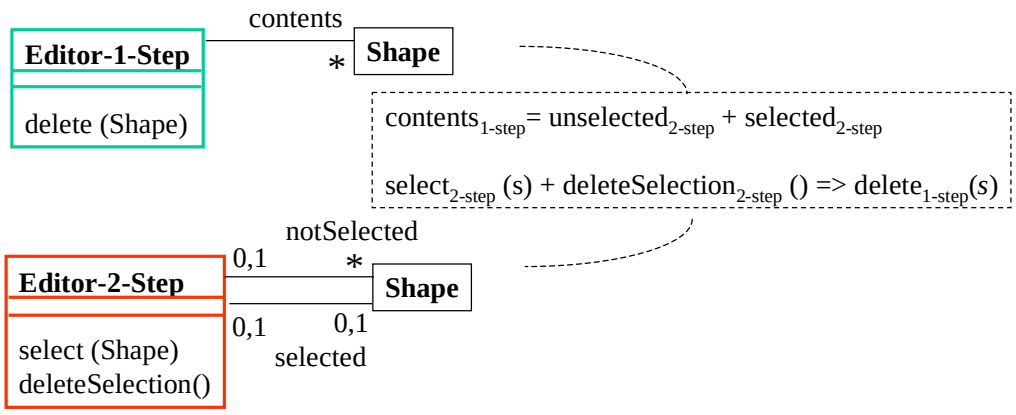
Editor-2-Step: later in the development cycle, we decide that shape deletion will not be supported as a single operation. Instead, the target shape must first be selected, and then deleted in a separate operation.

Clearly, the latter is a refinement of the former. Is it a sub-type?

Remember that a sub-type implies full substitutability i.e. any client who expected the super-type and its interface, should be able to use an object of the sub-type transparently i.e. without any change to the client.

The answer is *no*. Editor-2-step *does not support the protocol* of Editor-1-Step. Sure, it permits effectively the same things to be done, but the protocol is different. And, the *client has to be aware of this change*.

Refined Models and "Retrieval"



- Finer grained interactions induce a finer grained model
- Retrieve: Define abstract query in terms of refined model
  - Define refined sequences that achieve each abstract action
- Still, this is *not* a sub-type

We may even formalize the operations at each level of refinement:

Editor-1-Step: to express the *delete(shape)* operation, all we need is an attribute which describes the *contents* of the editor at any time. The result of *delete* is that the parameter shape is no longer in those contents.

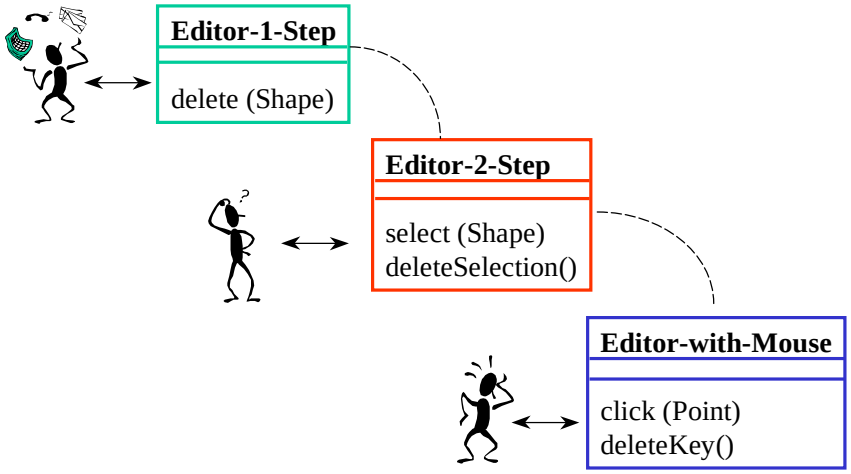
Editor-2-Step: to express *deleteSelection()* we need a concept of what was last selected, so we can say it is no longer in the contents. We add the attribute *selected* (drawn as as association here). We also have an attribute *notSelected*, indicating all other shapes. The *select(shape)* operation simply sets the *selected* attribute.

We can now formalize the operations in terms of these attributes.

We can even map ("*retrieve*") between the attributes. The *contents* attribute in the abstract version corresponds to the *unselected* + *selected* attributes in the detailed version (more properly, we would be manipulating sets here). We can hence argue that a sequence of *select* + *deleteSelection* achieves the same result as a single *deleteShape* in the abstract version.

Still, Editor-2-Step is ***not*** *a sub-type of Editor-1-Step!!*

Non-Subtypes: Varieties of Refinements



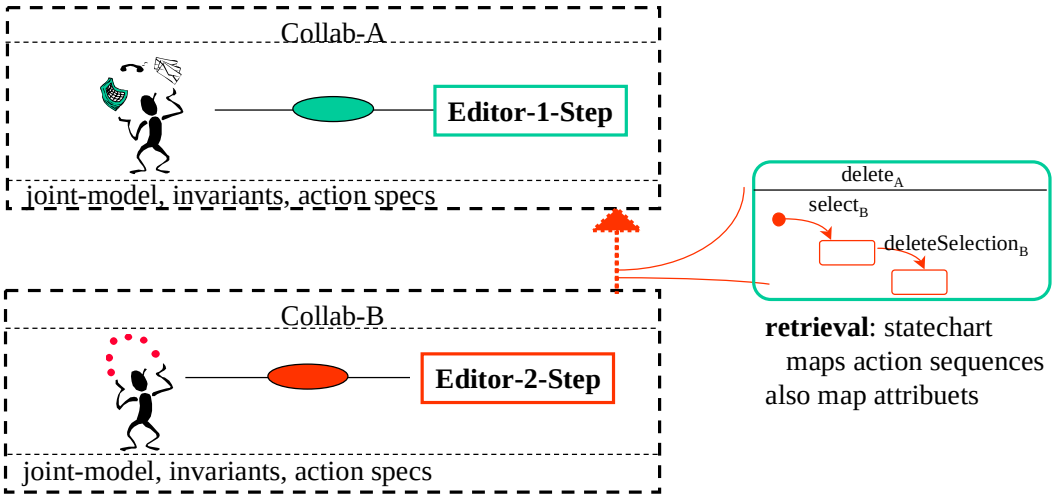
- Many common refinements do not create sub-types
  - Time granularity, signature, helper objects, ...

Here is that idea stretched a bit further, with yet another refinement. These are not editor sub-types.

In these examples, both the editor and the editing-client are affected by the refinement i.e. a multiple-object refinement.

What construct does Catalysis use to describe a multiple-object interaction?

Joint Refinement of Collaborations



- We refined the *joint interaction protocol*, or *Collaboration*
  - Both sides are affected by the refinement

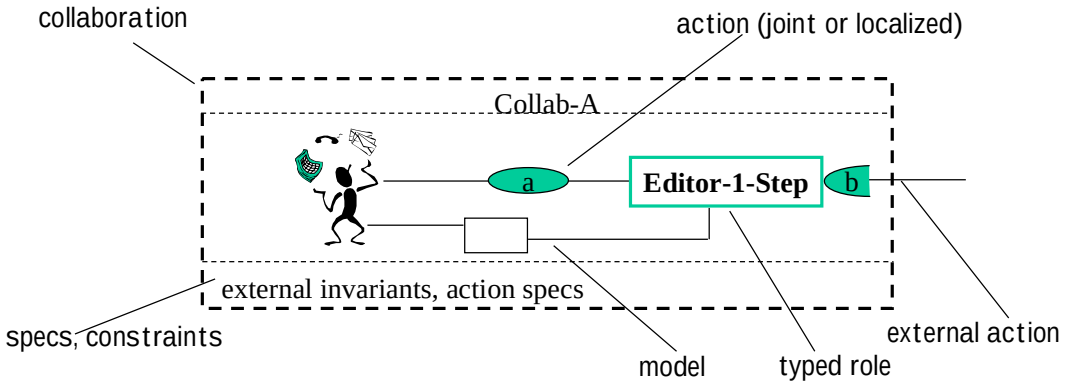
The unit being refined here is the *Collaboration.* Both parties are affected because the protocol is being refined.

The mapping from concrete to abstract must have two elements:

A mapping of attributes: for each abstract attribute, what (composition of) concrete attributes realizes that attribute? This can be described textually, formalized using OCL.

A mapping of actions: for each abstract action, what sequences of concrete actions realizes that action? This is often documented with a state-chart, or with a sequence diagram.

Collaboration

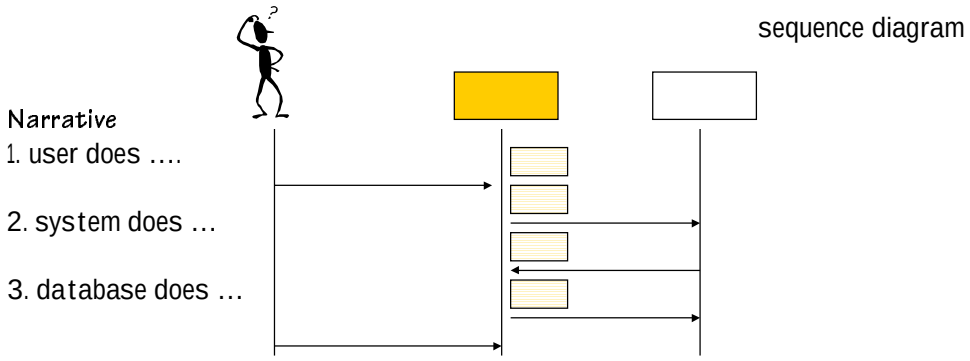


Collaboration:
A set of actions between typed objects playing certain roles, specified in terms of a common model. The actions themselves may be *joint* (not assigned to a particular type) or *localized* (responsibility assigned to a particular type), and may be *external* (not between collaborators, must maintain invariants) or *internal* (between collaborators, does not have to maintain invariants).

And here is how a collaboration is defined in Catalysis.

Scenario and Sequence Diagram

- A scenario is a trace through a collaboration
- Action pre/post attribute values are “snapshots”
- Snapshots conform/define type model
- Snapshot-pairs conform/define action specs
- Scenarios conform/define collaboration specs



A scenario in Catalysis is a trace through a collaboration. It is usually described in narrative form that tells a story about an interaction, and may have an accompanying *sequence diagram* to illustrate the interactions involved.

Scenarios and sequence diagrams are an important tool to flesh out requirements. However, it is not possible to exhaustively cover all scenarios. They must be accompanied with the more complete operation-specs and type model with invariants to cover the general cases.

Scenarios help with defining the type model:

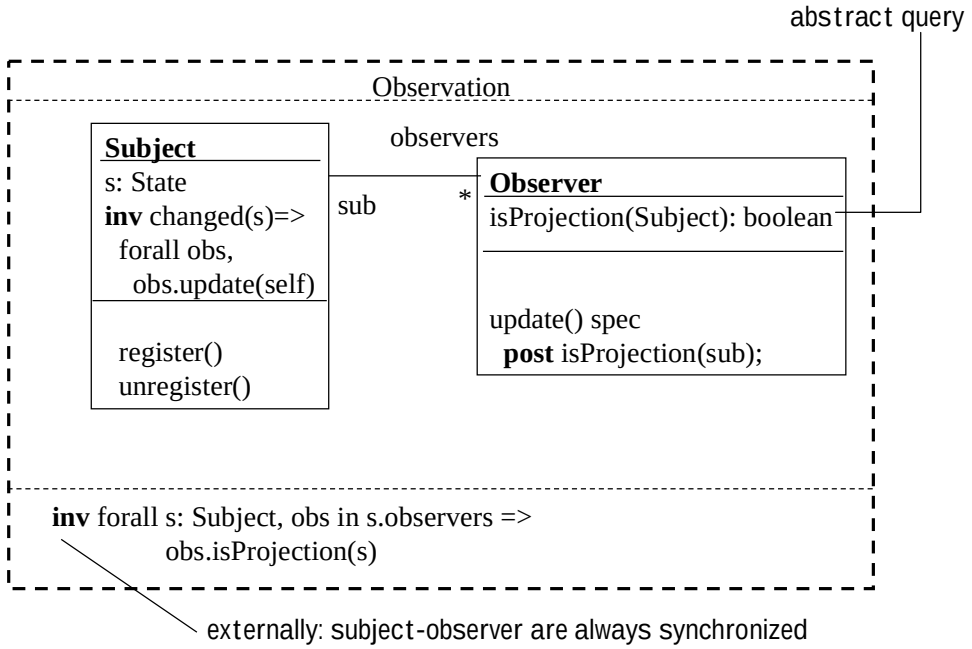
For each operation invocation on the system of interest, sketch out a before/after *snapshot* of objects (instances) to explain the effect of that operation invocation, assuming a certain prior state. The before/after configuration of objects in a snapshot pair describe the state change of the system; other outputs produced are also described.

The combination of all such *snapshots* must be covered by the type model of the system i.e. the type model generalizes all snapshots, admitting all legal ones, and prohibiting all illegal ones.

Scenarios help define operation specs:

For each operation invocation on the system of interest, the before/after *snapshot pair* describes the state change of the system, and other outputs are produced. The operation spec should cover all possible snapshot pairs i.e. it generalizes all operation invocations in all legal pre-states.

Subject-Observer Collaboration



Many design patterns are best understood as abstract collaborations.

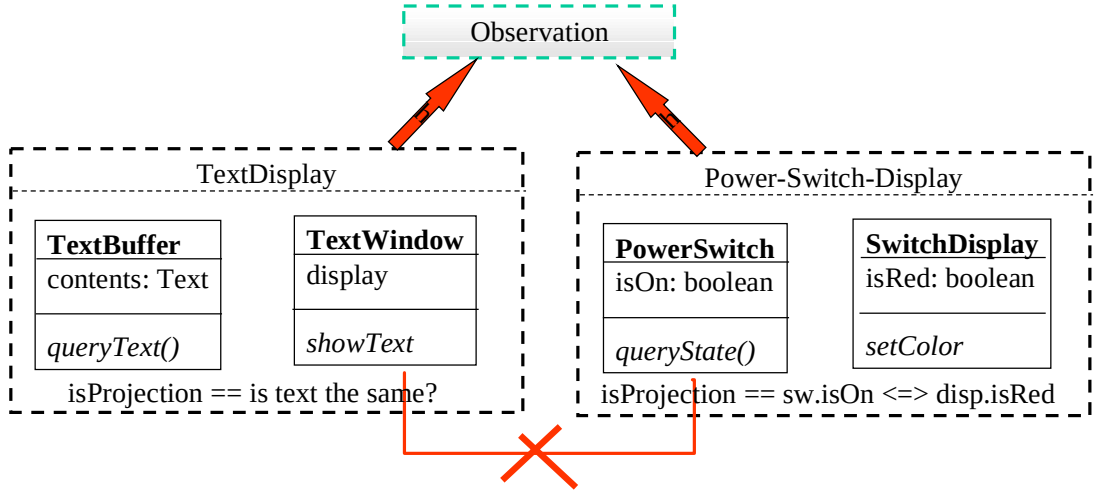
Design patterns, though abstract, should still be expressed with some precision.

This slide shows an example how:

- Every subject has a state attribute, **s**. We don't really know much about the type of **s**, except that something special happens when **s** changes.
- Whenever **s** changes, every observer of that subject has been **updated**
- Every observer has its own concept of being "in sync with" its subject. We don't know what this definition will be for different observers, so we abstract it with a boolean attribute **isProject (Subject)**.
- Every observer will provide an **update** method. We don't know exactly what each update implementation will do, but we do know that it must achieve whatever is the corresponding definition of **isProjection**.

When I apply this collaboration to a particular problem domain, with unique kinds of subjects and observers, am I creating sub-types of subject? Sub-types of observer?

"Specializing" Subject-Observer



- Do not confuse this with subtype or subclass
  - The entire family of related types (playing roles) is being specialized
  - Will be addressed by *"frameworks"*

Our answer is *No*.

Supposed we had two applications of this pattern:

> A text window displays pixels that must be kept in sync with the Text contents of a text buffer.
>
> A switch display shows a color that is kept in sync with the corresponding state of a switch.

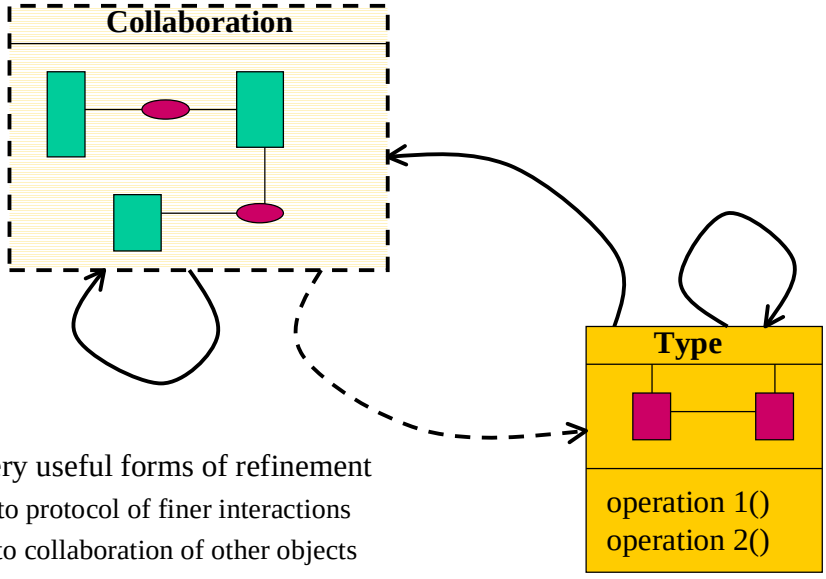
Are the two observers interchangeable? Could you use the text window unchanged as an observer for the power switch? How about the switch display to observe the text buffer?

Clearly, text window and switch display are *not* subtypes of Observer; nor are text buffer and power switch subtypes of subject.

Instead, the *pairings* of these types constitutes a refinement of the subject-observer collaboration i.e. both parts must be mutually compatible when refined.

We will show later how Catalysis frameworks provide an even more powerful way to deal with this.

Forms of Conformance and Refinement



- Several very useful forms of refinement
  - Action to protocol of finer interactions
  - Object to collaboration of other objects
  - Type and model to class and instance variables
  - Signature refinement

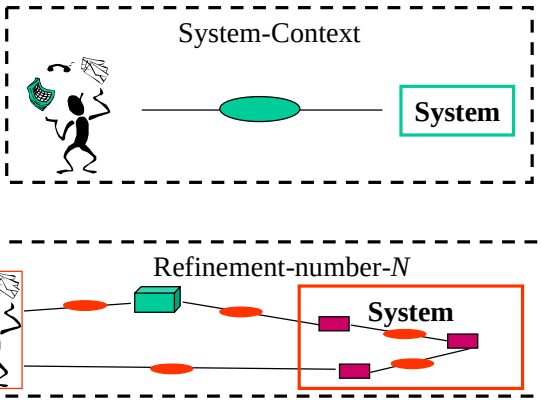
Catalysis provides a core set of refinements, that can once again be combined and composed in very many ways. These include:

Collaboration refinement: the protocol or signatures are being refined in a way that affects multiple participants in the collaboration.

Type refinement: a single object type can be refined by a more detailed model, in a way that does not affect clients at all. e.g. a class implementing a type.

Type decomposition: a single object type is designed as a collaboration of finer grained object, in a way that does not affect clients at all.

From Use-cases to Code (and Back)



- Collaborations and refinement provide full traceability
  - Use-cases at the level of system and user-tasks
  - Refinement of interaction granularity, external and internal roles
- Clear semantics for use-case development, business to code

 70

The foundation of *Refinement* and *Collaborations* provides Catalysis with a well-founded basis for development using *Use-Cases*. Use-cases are a nice technique for focusing on user-tasks, and refining these tasks down to code. Unfortunately, almost all accounts of use-cases suffer from a lack of precise definition and meaning, and even use-case "experts" are often hard-pressed to justify why they do certain things, and what the real meanings of "extends" and "uses" are in structuring use-cases.

In Catalysis, a use-case starts off being a single *abstract action.* It has some participants (possibly more than 2), and has a post-condition. There may be many possible sequences of interactions that realize that post-condition, and they are not an intrinsic part of this *abstract action*. There are a corresponding set of attributes that support the expression of this post-condition.

The abstract action is subject to refinement. We may refine it by:

Refining the interaction protocol, while keeping the same participants. The attributes are necessarily refined, and map to the abstract attributes. Sequences of concrete actions map to abstract ones.

Refining the participants themselves. What was treated as one "system" object in the abstract action, may actually be comprised of several smaller objects, without any "real" object for the system itself. The corresponding attributes are changed, and map to the abstract attributes. Typically, the actions are also refined at the same time.

Introducing new participants and helper objects. These act as intermediaries, or provide new aspects to the actions that may not have been a required part of the original abstract action spec.

Of course, since use-cases fit into a more general framework of actions, collaborations, and refinement, they can be structured with very well defined interpretations of "extends" and "uses".

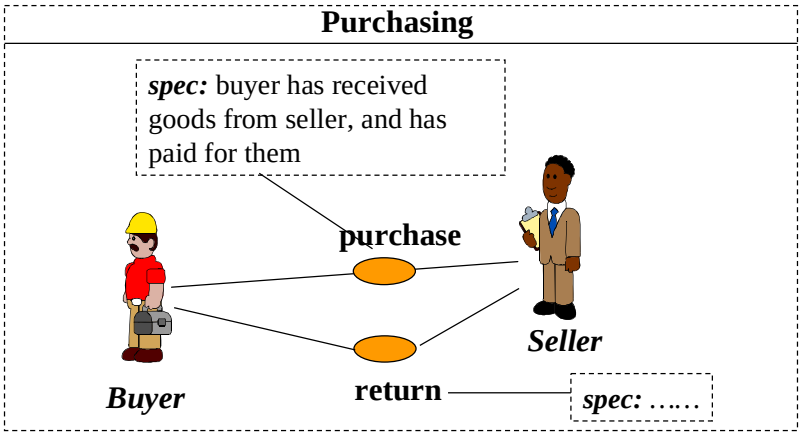
Outline

- Method Overview
- Component Specification - Types
- Component Design - Collaborations
- Component Architectures
- Refinements
- **Business Example and Development Process**
- Frameworks

71

This section will step out of the realm of software, and show how the concepts in Catalysis are equally applicable to the modeling of businesses.

Collaboration - Business Model

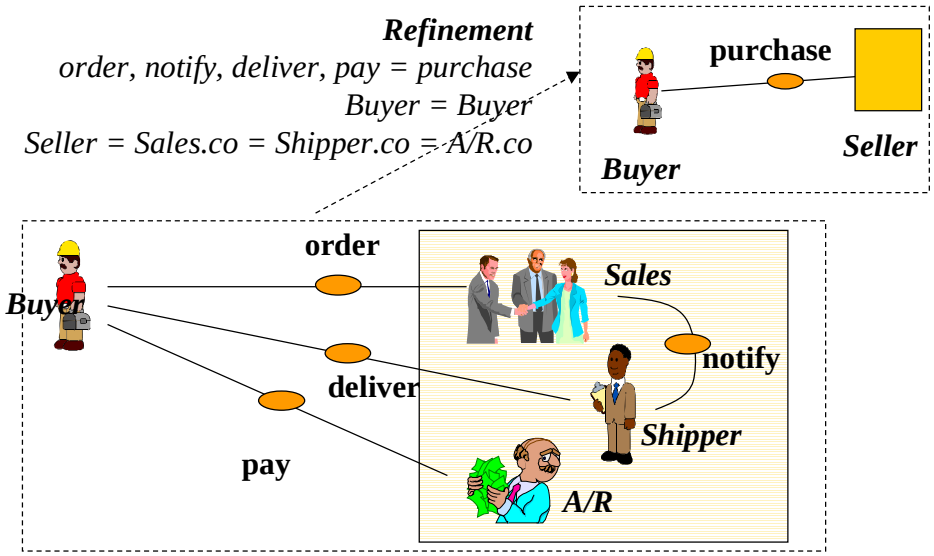


- Collaboration: a set of related actions between objects
- Actions and objects can be business processes and roles
- Each action has a specification and model of types

A collaboration applies easily to any business process that can be described as an abstract action.

This collaboration shows two abstract actions, *purchase* and *return*, involving two roles/types: *buyer*, *seller*. These actions have a spec, and some attributes on buyer, seller, (and product, item, money, etc.) to support the action specification.

Business Refinement



- A relation and mapping from a detailed to an abstract description of the same phenomenon

And the idea of refinement nicely supports the concept of business process detailing or decomposition.

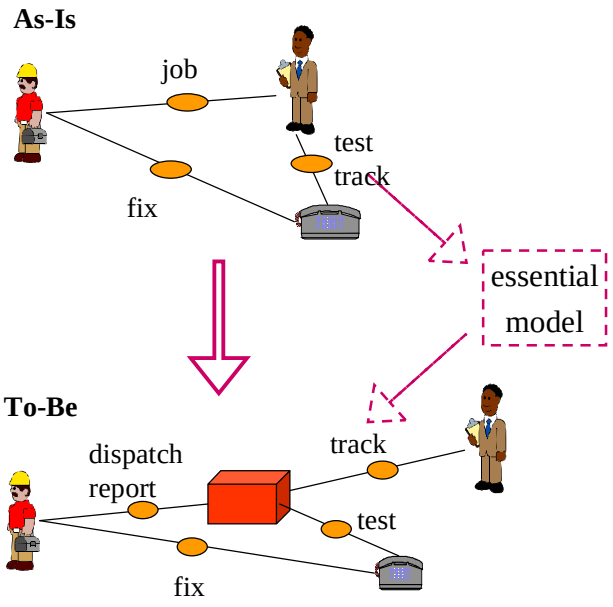
This refinement combines two things:

Action refinement: an abstract *purchase* action is refined in a sequence of order, deliver, pay.

Type refinement: an abstract *seller* type is refined to the *sales, shipper, and a/r* roles

There are more detailed attributes in the refined model, and a mapping from the refined to the abstract in terms of attributes and action sequences.

Business Design: As-is and To-be



- Analyze existing interaction
- Extract essential behavior
- Do a re-design (refinement)
- Framework for ***use-cases***
- Many business implications
  - performance
  - new functionality
  - cycle-time
  - error rate
  - costs
  - phased transition plan

In face, the very buzz-phrase *Business Process Re-engineering* also has a well-defined meaning in Catalysis. The process of business (re)design consists of:

Build an *as-is* model of the business, at least in the areas of suspected problems

Build a more abstract model of the real business purpose being served, abstracting away some specifics of how it is done today

(Re)refine the abstract model to a better-engineered business design, perhaps with some tasks cut out, some tasks modified, and some tasks automated.

This is often a collaboration re-design, since many players are typically affected.

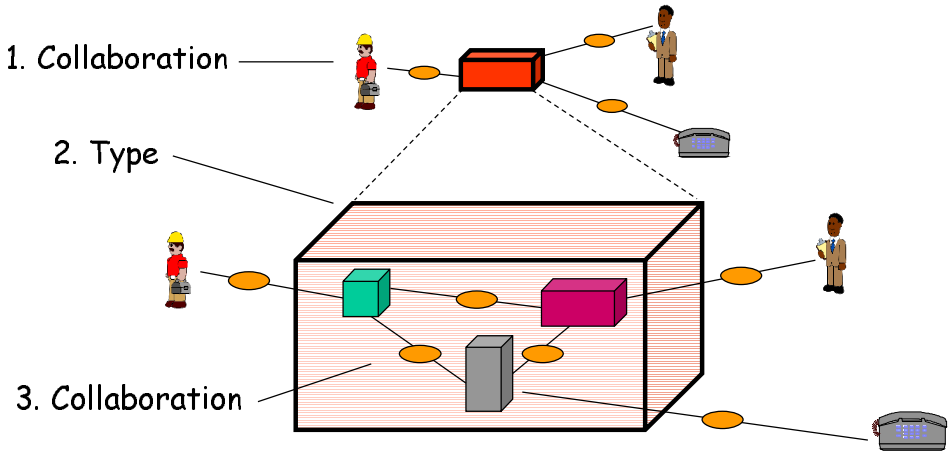
This interpretation carries over to most forms of re-engineering.

First, study the existing design and identify places to change or improve

Second, abstract the existing design to the essential task being performed

Third, (re)refine the abstract description to the new re-engineered design

Recursive Process - Software or "World"



- A software system is an "object" in the business context
- Its internal design will be other collaborating objects

The model of collaboration, type, and refinement carry us from business or "world" models to code.

A software system itself plays a role in business collaborations, with external "actors"

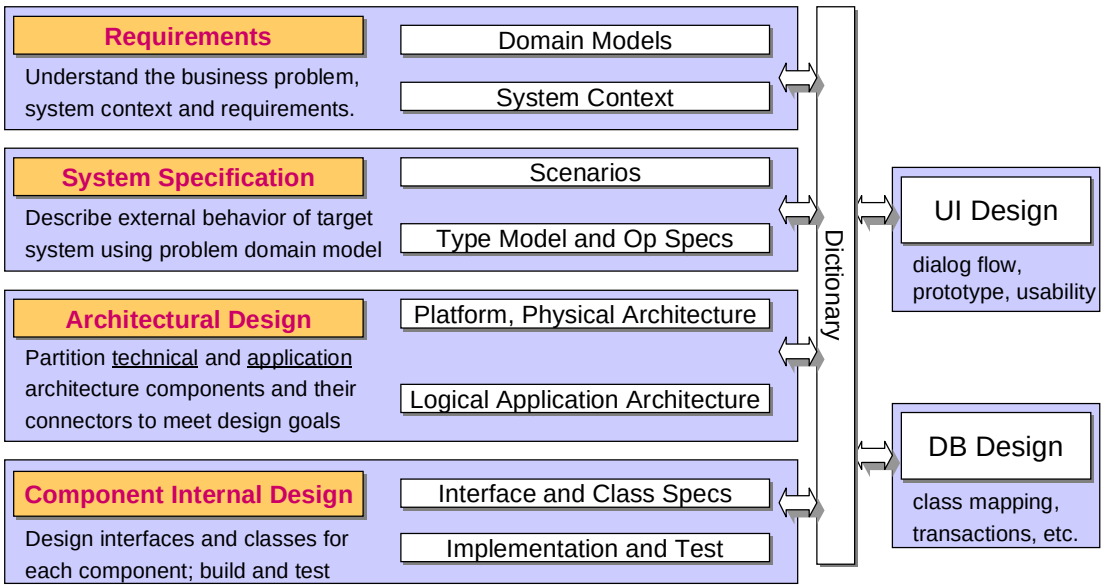
It can be characterized separately by each collaboration it participates in.

It can be specified as a (large) type

Its insides can be designed as a collaboration of finer-grained components

UML Development Process with Catalysis

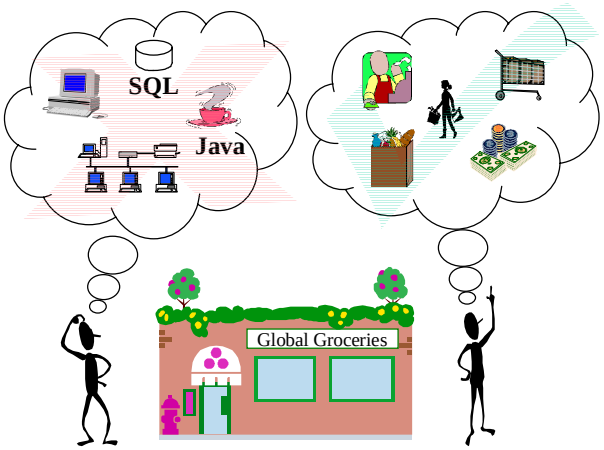
UML = Unified Modeling Language, standard notation for OO design



These next few slides outline a strawman development process using Catalysis. Details of the process are discussed elsewhere.

Focus on the Problem Domain

External models should reflect the customer's view of the problem domain, *not* the programmer's view.



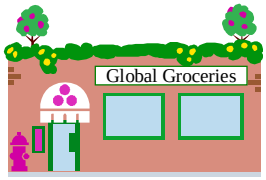
A problem domain focus helps to ensure continuity between the software and the real world.

Continuity makes it easier to
- verify model with customer
- train new developers
- estimate maintenance effort
- identify sources of defects

Any external model should reflect the client's view. The concept of *refinement* lets us map between internal views and the view that is most natural for a client to understand.

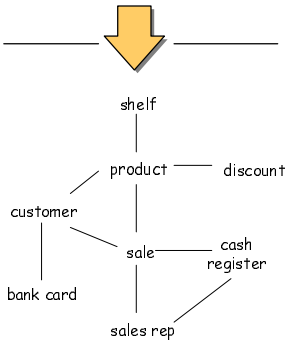
Requirements | System Specification | Architectural Design | Internal Design

Problem Domain Analysis



Domain Knowledge

Problem Domain Analysis - That part of the development process dedicated to developing an understanding of the environment in which a target system will reside as well as the systems role in that environment.

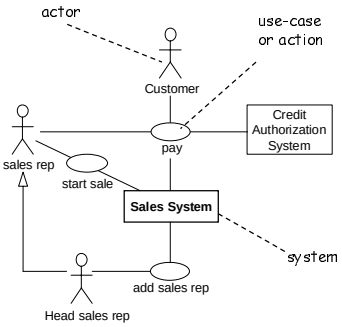


Mind Map

Entities
sales rep…
customer…
discount ...
product…
sale…

Actions
add sales rep…
back-order product...
start sale…

Dictionary

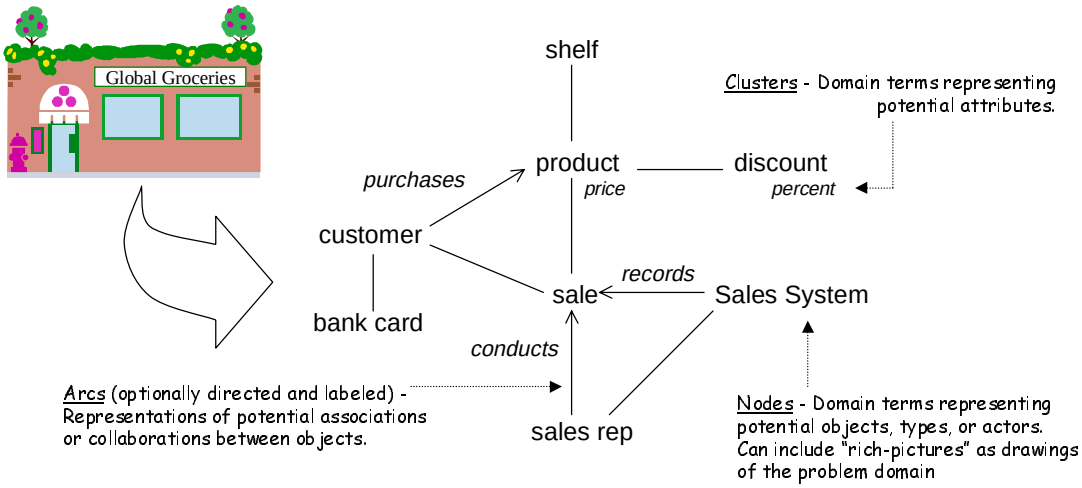


System Context

Problem domain analysis is the process of understanding the context for a development effort, and starting to define a solution to some identified business opportunity or problem.

Mind Map: Informal Problem Domain Model

Mind Map - An informal structured representation of a problem domain
Not a stored data model!!



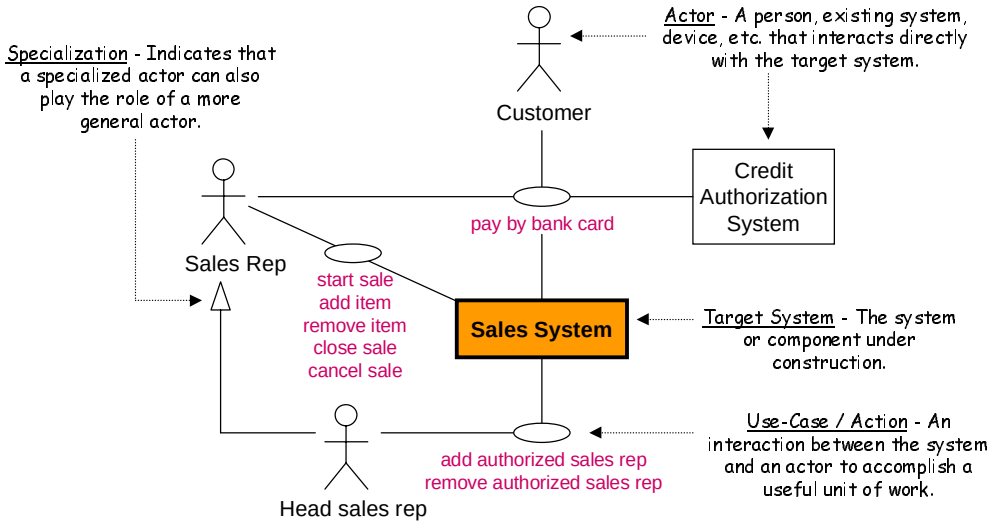
Can be formalized if appropriate

A mind-map is a very informal description, and may be enhanced with other informal tools like storyboard, rich pictures, etc.

A problem domain model can also be a formal collaboration model of the domain itself, specifying the object types (human, organizational, machine), the actions they participate in, specifications of those actions in terms of a type model of all objects involved, etc. This model can use all the formal tools of Catalysis, including refinement and frameworks.

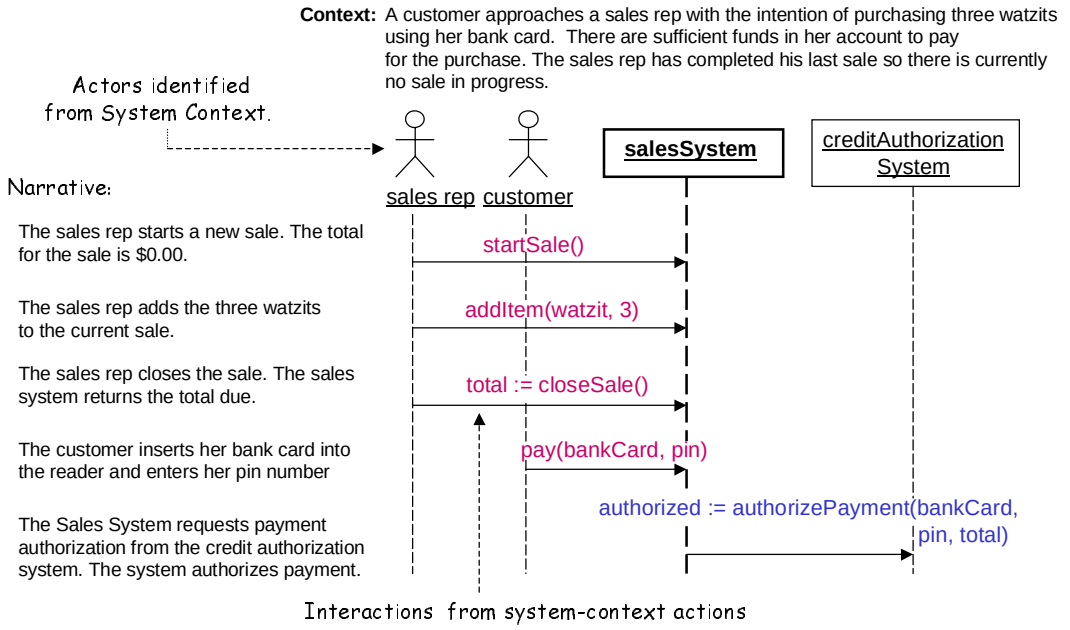
System Context -- a Collaboration

System Context Model - A structured representation of the collaborations that take place between a system and objects in its surrounding environment (context).



This system context shows all the external objects that will interact with the system of interest, and describes, at a high level, what their interactions will be.

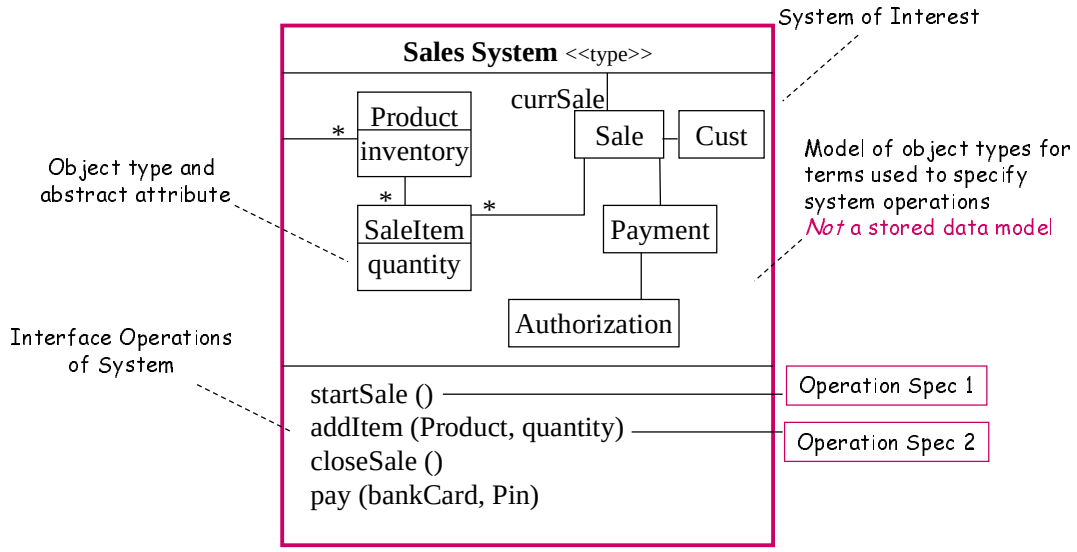
Scenario of Use



A scenarios depicts a specific trace through a collaboration.

Requirements | **System Specification** | Architectural Design | Internal Design

Type Model and Operation Specs

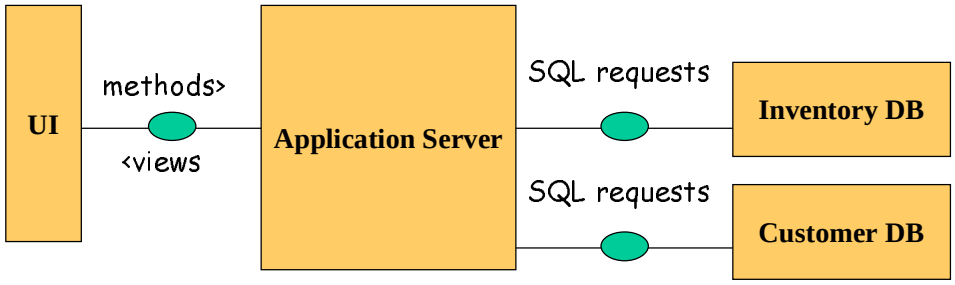


Note: Does not as yet commit to operations on individual classes inside system
Internal component partitioning and class design not decided yet.

This model provides the essential specification of the system itself, described as though it were a single object. Of course, the description is in terms of object types, and their attributes, associations, etc. from the problem domain itself I.e. those parts of the context this system will need to be aware of in some form.

Requirements System Specification **Architectural Design** Internal Design

Architecture: Platform and Physical



- 3-Tier Architecture
  - Thin Client: User interface, dialog flow logic
  - Application Server: Business objects and rules
  - Oracle Database: Persistent data storage
- Must explicitly evaluate technical architecture factors
  - Scalability, performance, thruput, seats, platform, clustering...

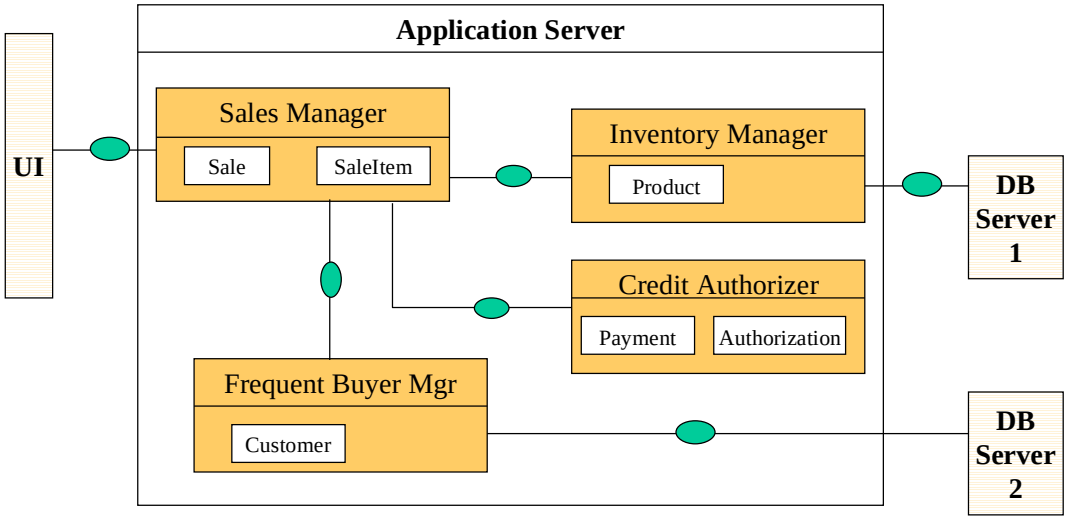
Note: 3-Tier could also be a purely logical partition of presentation, business objects, store

Technical and physical architecture is often neglected in projects. It should be begun early, and should be driven by factors such as scalability, performance and other non-functional goals, thruput, database performance, clustering capabilities of the target platforms, etc.

The architecture should be validated by some combination of past experience, simulation, and architectural prototypes.

Requirements System Specification **Architectural Design** Internal Design

Architecture: Application Components

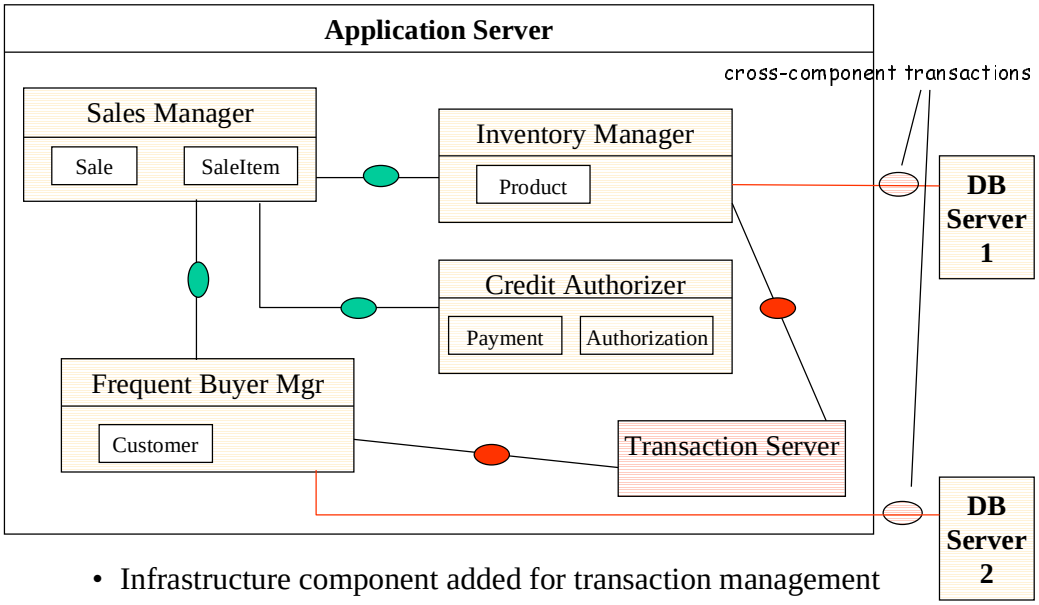


- Application components partitioned and interactions designed
  - Partition based on ease of change, re-use, buy Vs. build, etc.
- Original domain types split across components, some partly shared

The application architecture is somewhat independent of the technical architecture. It consists of a partition of the application behavior into separate components, based on criteria such as re-use, parallel development, localization of variable parts, etc.

The problem domain types will be spread across these components, with some links between them realized in a form that depends on the technical architecture chosen e.g. direct pointers, OODB shared object references. Sometimes, the object types from analysis will appear in a somewhat generalized form in these components e.g. a concrete kind of problem domain resource may appear as a generic *Resource* in a resource-manager component.

Architecture: Infrastructure Components



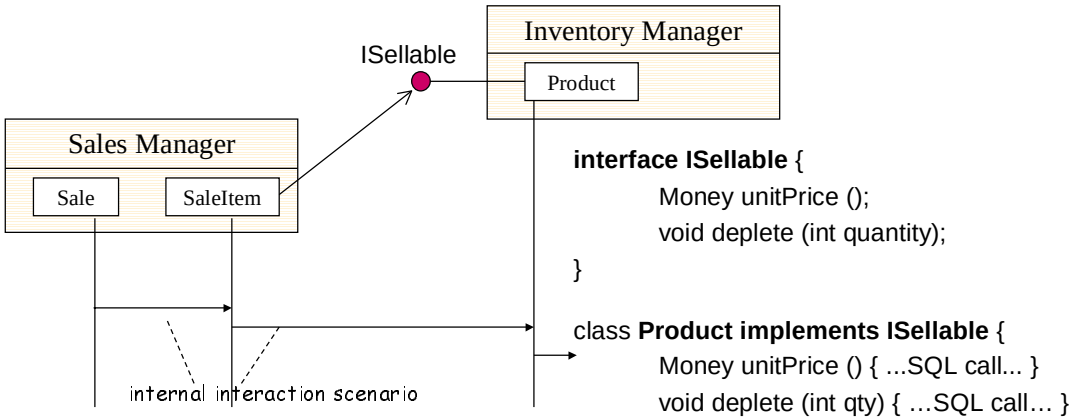
- Infrastructure component added for transaction management
  - Appropriate application components "linked" to it

In addition to the application architecture, we must also consider explicit infrastructure components, particularly for distributed enterprise systems. e.g. the middle tier of a multi-tier system will frequently be responsible for co-ordinating distributed transactions, managing various resources across multiple users, etc.

The design should leverage of standard architectural components for these services where possible e.g. using CORBA transaction or event services, or using Microsoft's Transaction Server.

Component Internal Class Design

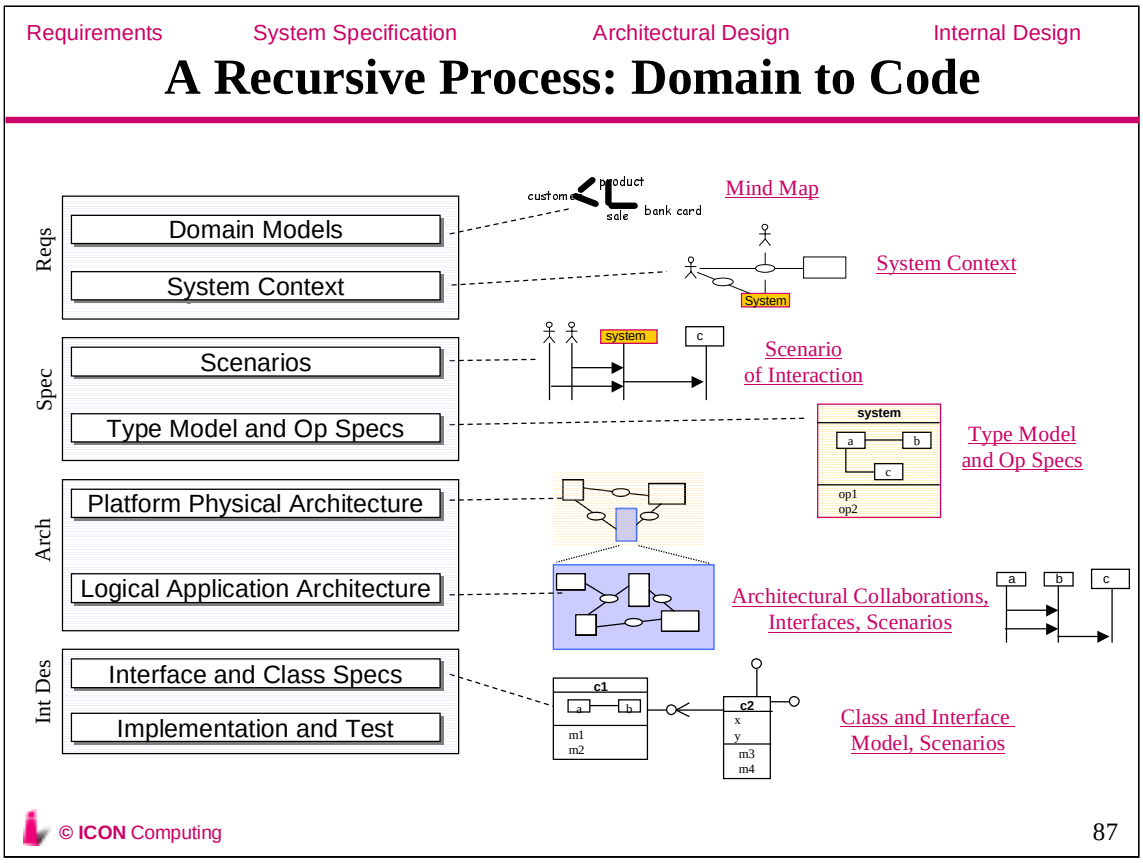
- Design exposed interfaces of objects visible across components
- Design collaborations at exposed interfaces
- Design collaborations triggered from exposed interfaces
- Design classes to implement exposed interfaces and interactions




```
interface ISellable {
        Money unitPrice ();
        void deplete (int quantity);
}

class Product implements ISellable {
        Money unitPrice () { ...SQL call... }
        void deplete (int qty) { …SQL call… }
}
```


Each application component has to be designed and implemented internally as a set of classes that implement the interfaces required externally, as well as realize the internal interactions required by the original OOA.

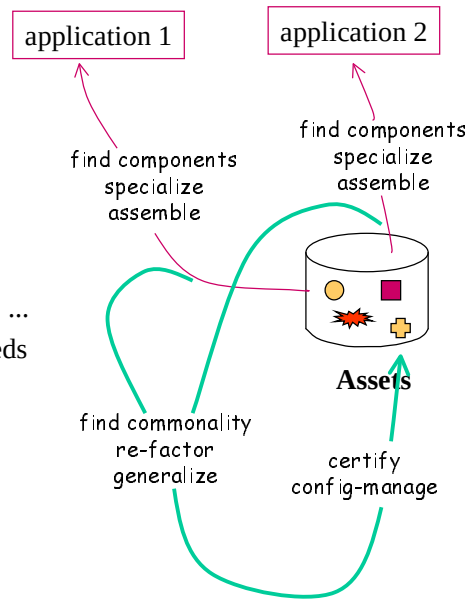
The resulting design can (and should) be mapped to the analysis level description of object types. e.g. a design review must inspect this mapping to justify design choices made, and the correctness of the design.



The entire process is recursive, focused on the collaborations between components or objects. The concepts of type models, collaborations, refinement, frameworks and patterns can be applied at all levels and across the entire process.

Two Distinct Development Cycles

- Applications
  - rapidly built solutions
  - specialize and assemble Assets
- Assets = Reusable components
  - includes source, executable, designs, kits, ...
  - generalized from multiple application needs
  - not dreamt-up re-use



The artifacts of design and the relationship between them are never developed in a completely top-down manner. In particular, when developing with components, there are two distinct development cycles at work. Recognizing these cycles and explicitly planning for them helps maintain an understandable development process, despite the inevitable unpredictability and opportunistic nature of development.

Development Cycle for Applications

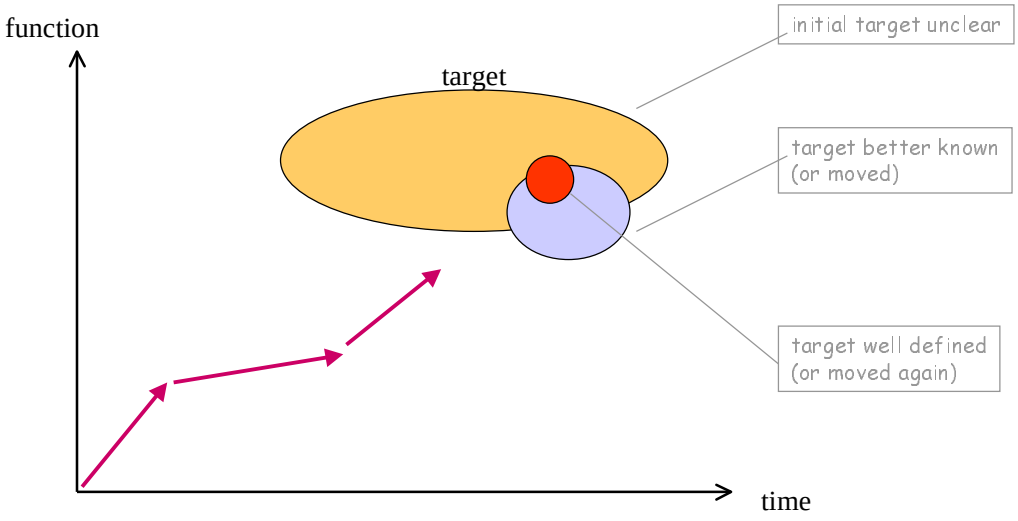
- Developing Applications
  - A managed RAD method is usually the most effective
    - Rapid requirements modeling and prototyping
    - Short delivery cycles
    - Close user involvement

If I had to live my life again, I'd make the same mistakes, only sooner

-- Tallulah Bankhead (1903-1968)

Many applications should follow a RAD cycle (although this may not be suitable for some special cases).

Why Iterative and Incremental?



- Not everything needed is known up front (or at the “end”)
- Not everything known is needed up front (or at the “end”)
- Frequent iterative and incremental delivery helps function and timeliness

A RAD approach, in which decisions are iteratively made and refined, and the application is delivered in stages of incremental functionality, decreases the risk of unknown of changing requirements, *provided the development is well managed*.

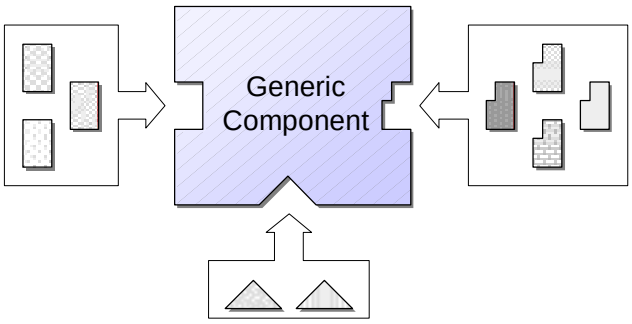
Development Cycle for Components

- Developing Components for re-use justifies investment
  - Well specified and documented interfaces
  - Robust against misuse; does not just crash
  - Generalized for variation points with parameters, “plug-points”
  - Associated packaging with tests, specs, default “plug-ins”, etc.
  - Carefully chosen architecture for “kit” of components
  - More careful certification and configuration management

Reusable components justify a more thorough specification approach.

Re-usable Assets and Variation Points

Variation Point - A location at which a generic component may be specialized for use in a particular application.



A generic component may come bundled with a set of pre-built variants for some or all of its variation points or the host application may provide its own variants. A variation point may have a default variant.

And generalizing a component to be re-usable takes special care.

Golden Rule of Design

Design for change.

- Distinct Categories of Change
  - Modification
  - Extension
  - Reuse

Open Closed Principle - Ideally a method, class, or component is open to extension but closed to modification.

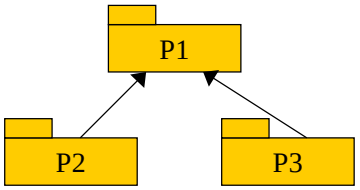
Outline

- Method Overview
- Component Specification - Types
- Component Design - Collaborations
- Component Architectures
- Refinements
- Business Example
- **Frameworks**

And that brings us to *Frameworks*, the Catalysis construct for characterizing common patterns that occur across types, collaborations, and refinements, while permitting adaptation and “plugging-in” to those patterns.

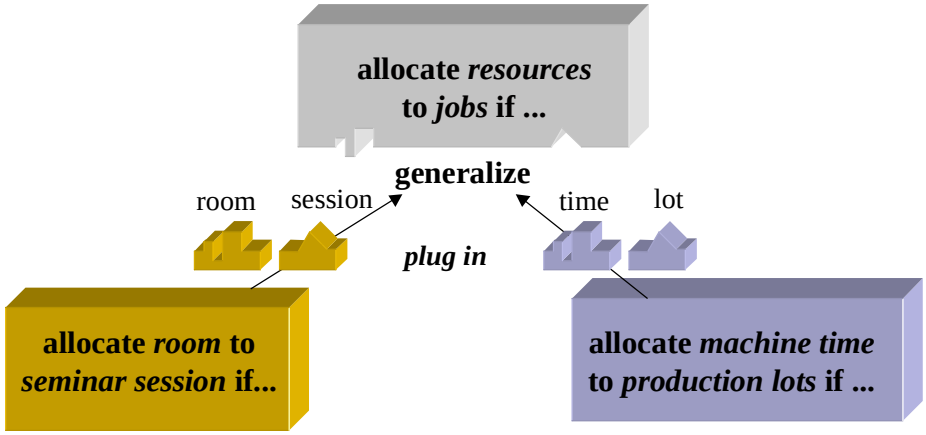
Exploiting Packages

- A package is unit of “knowledge”
  - a unit of work, CM, versioning, etc.
  - can import other packages
  - a “closed world” (modulo imports)



- Essential to separate refinements e.g. interface vs. implementation
  - implementation imports interfaces it implements and uses (with their behavior specs, to the extend implementation depends on them)
  - each package constitutes a component
- Can say different things about the “same” entity in P1, P2, P3
  - better ways of structuring, project planning, composing
- Strong basis in formal methods: *traits* and *theories*
  - formalisms made approachable

Frameworks - *Generic Components*



- A generic model / design / implementation component that
  - Defines the broad generic structure and behavior
  - Provides *plug-points* for adaptation

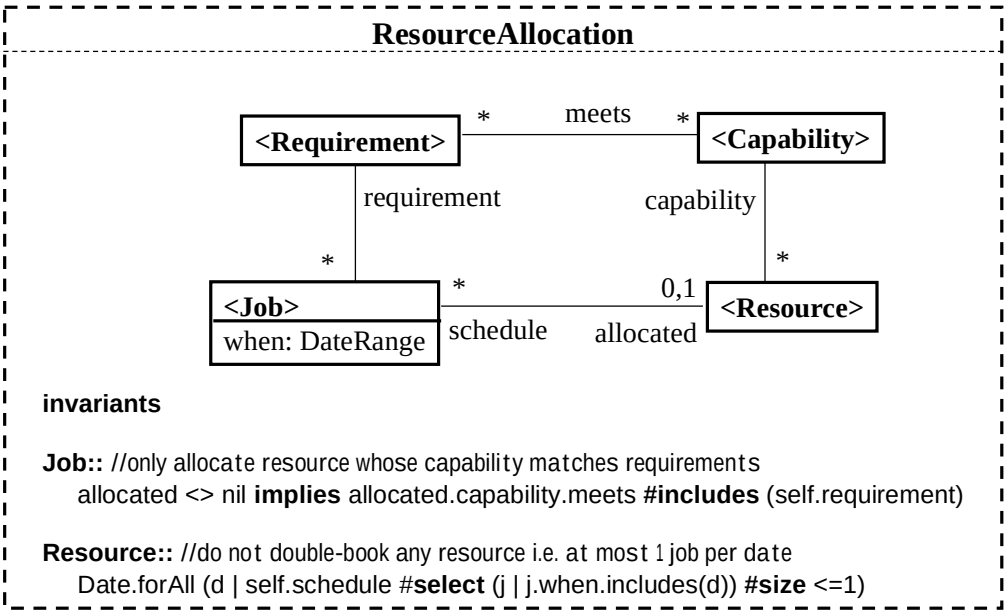
Consider the two concrete applications shown, one for seminar scheduling, and the other for factory floor automation.

Could we make both pieces of requirements look the same, by abstracting out some differences?

Sure. You end up with a description of a generic resource-allocation problem.

Of course, in Catalysis we do not want to lose precision just because we have become more abstract.

Resource Allocation Framework

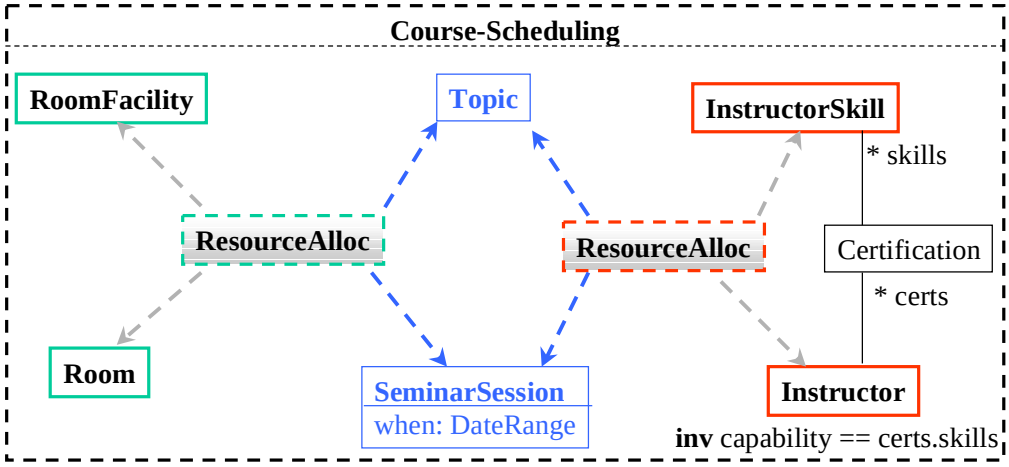


So we build a proper type model to express what we want to say.

We can even formalize, if appropriate, the operations specifications and invariants that apply to the generic problem of resource allocation.

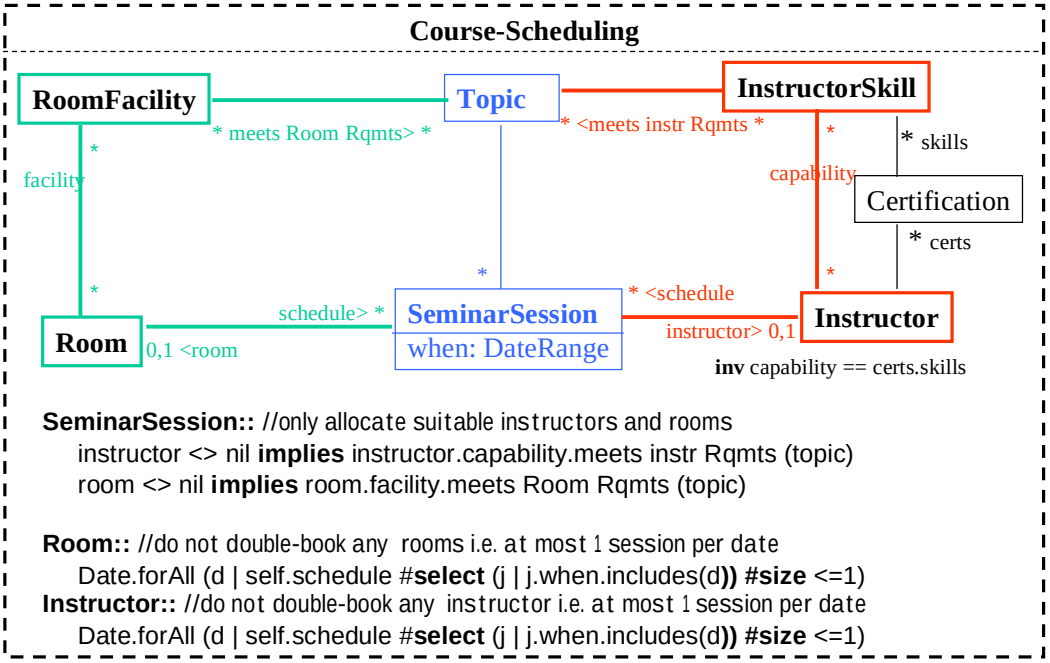
"Applying" a Modeling Framework

- **"Apply"** resource-allocation **twice** to course scheduling
  - Each application substitutes different *resource, capability, etc.*
  - Both apply to the same *job*: Seminar Session



We can then *apply* this resource-allocation framework to a particular problem domain. In this case, to seminar scheduling where we allocate rooms and instructors for each seminar.

The "Model" is Automatically Generated

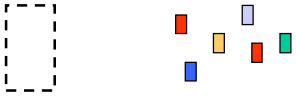


By applying the framework, and making the appropriate mappings, we have implicit;y defined the entire model shown above. An intelligent tool would be able to easily switch between the summary and detailed views.

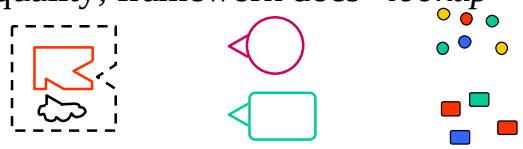
This simple example underscores a very powerful capability in Catalysis for abstracting and re-using patterns of models, specifications, or designs in different contexts in a much more effective manner than just *cut-and-paste*.

What is a Framework (in *Catalysis*)?

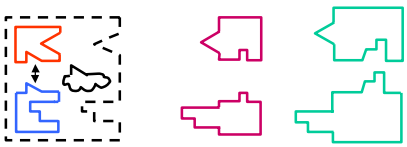
A *type* defines properties of a set of objects
- *Instructor, SeminarSession*



A *framework* defines properties of a *set* of *suitable types*
- *DictionaryEntry:* any type with equality, framework does "*lookup*"

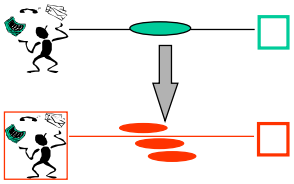


A *multi-type framework* defines properties of a set of
families of suitable and *mutually compatible* types
- Allocate: *Resource-Job*

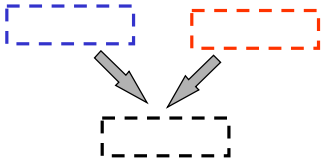


Frameworks: Two Supporting Dimensions

- Frameworks can be described at different levels of *refinement*



- Frameworks themselves are *composed* of smaller frameworks

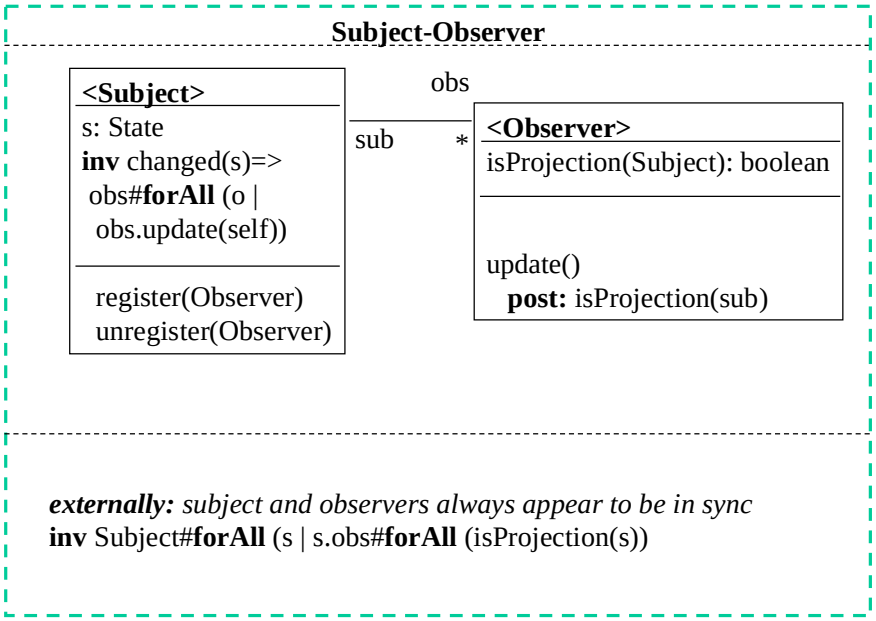


Frameworks in Catalysis are more flexible and expressive, and hence more practical for complex problems, because of two other capabilities that they can leverage:

Like any other models, frameworks can be described at many levels of refinement.

Frameworks themselves can be composed of other frameworks.

Design Patterns as Frameworks



Subject-Observer really is such a framework. When we "apply" it, we need to substitute for appropriate parts of this frameworks.

Specifically, this frameworks is expressed in terms of types, not classes and inheritance.

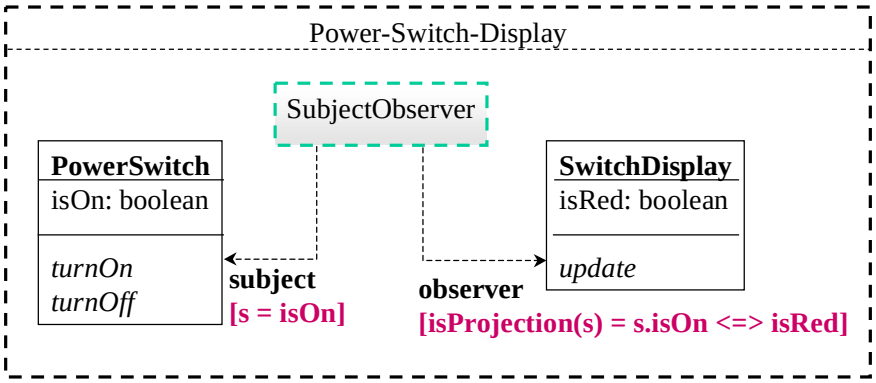
We have one type *Subject,* and every subject has a *state*, called **s**. We do not know the type of this state attribute, as it may be very different from one application of this framework to another. It may range from a string, to the length of an array, to a complex type. However, in all cases, there is one underlying requirement on this state type -- whenever it *changes*, every observer must be updated. I.e. we need a definition for *changed* for any type that corresponds to *State* in any application of this framework.

Similarly, any subject has *observers* that can be updated when the subject changes. Once again, we do not know how exactly any observer will update itself -- e.g. it can range from changing a color on a screen from green to red, or updating some cached value, or generating further notifications. As before, we abstract out these differences *without losing any of the precision we need.*

We have introduced a parameterized boolean attribute *isProjection(subject)* on the observer. When we apply this framework, each observer must come with some corresponding definition of *isProjection*. Moreover, when that observer's *update* method has been invoked, it must guarantee that the observer is a projection of that subject.

This framework also illlustrates an additional feature of Catalysis. A collaboration also defines a unit of scoping for actions. Some actions are *internal* to the collaboration, others are *external*. We can now define *invariants* that apply to all external actions, such as those actions not listed in the collaboration. However, the internal operations, such as *update*, may be invoked even when these invariants do not hold; in fact, in this case, the internal actions are specifically designed to restore those invariants.

Applying Design Patterns



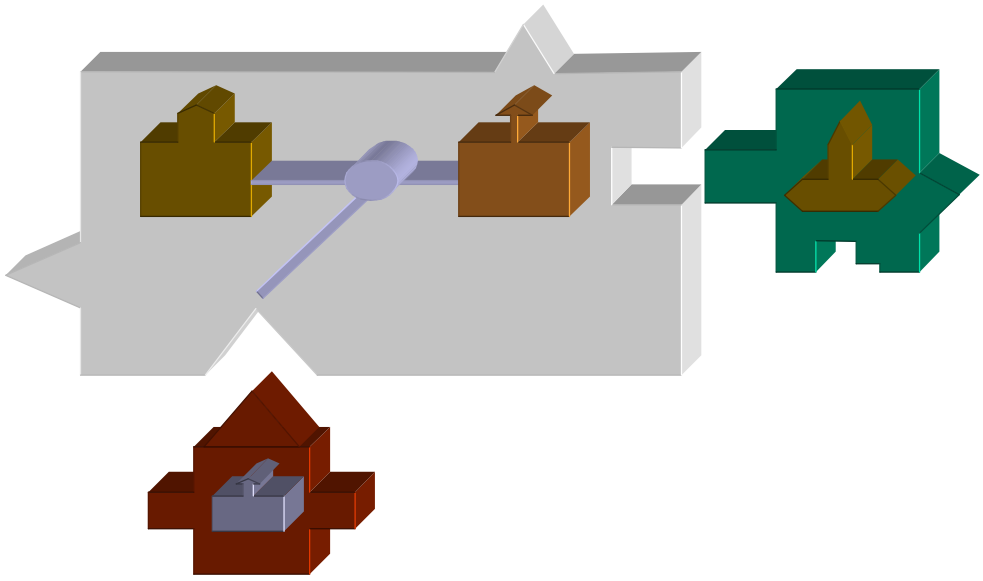
- The instantiation defines mappings of types, queries, actions
  - Needed to generate the instantiation, and for the “retrieval”

And here is what it would look like upon substitution.

We have made our PowerSwitch play the role of a *subject* with respect to a SwitchDisplay that is its *observer*.

We have also correspondingly substituted the *isOn* attribute for the abstract state of a subject, and a simple rule for the *isProjection* attribute of the observer.

The Holy Grail of Software Development



- Assemble independently delivered components at any *level* and *binding time*

Catalysis makes use of frameworks for requirements, designs, architectures, user-interface standards, refinements, use-cases, and more. This is just a fancy graphic for suggesting the prevalent theme of composition and "plugging-in" at all levels of development.

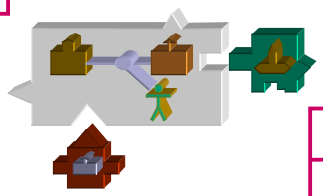
The underlying technologies of polymorphism, dynamic binding, and frameworks with substitution, take us a few steps closer to that *holy-grail* of software development:

> *Assemble independently delivered components of any size or granularity, at any level of development, and at any binding time from conception to run-time.*

Example Frameworks at All Levels

Business Models
Barter
Trader
Authorizer

Domain Models
Resource Allocation
Account Settlement
User-Interface Patterns



Design Patterns
Subject-Observer
2-Way Link
Cache
Moving Window
Data Normalization

Fundamentals
Total Ordering
Groups
Range
Descriptors

- Constructive approach to modeling and design with full traceability
- Libraries and commerce of frameworks of models, designs, and code

Catalysis frameworks span a wide range, from business to academic-sounding fundamentals:

Business models:

What does it mean to *barter*? It means two parties swap quantities of two dissimilar items, with some notion of equality of value between the item types.

What is a *trader*? A party that mediates in a *barter*, and adds on a couple of additional barters for the service.

What is *authorization*? ….

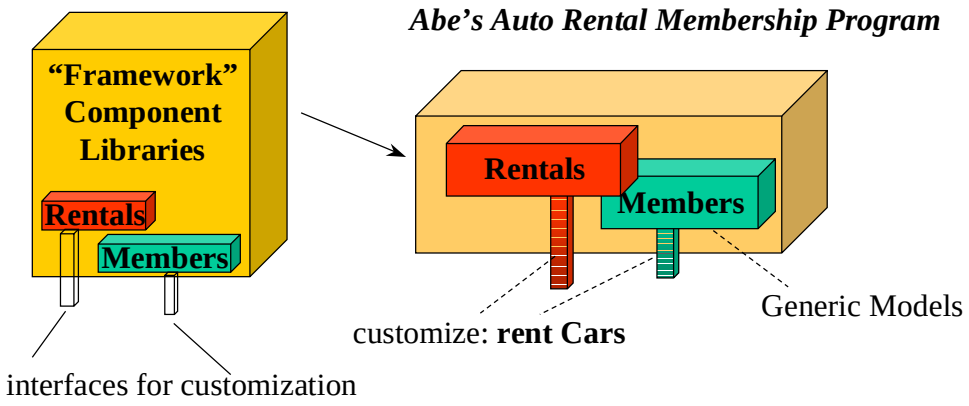
..down to fundamentals:

What is a *total ordering*? This framework applies to numbers, money, time, latitudes, ...

What is a *range*? This frameworks applies to time, money, numbers, …

…and so on.

Component-Based Modeling and Specs



- Build models, specs, and implementations from *generic component libraries* by customizing and composing

Our vision of component-based development with frameworks calls for the rapid composition of components for models, design, and code, with adaptation and customization of frameworks. Much of modeling or design work should be done by such composition, as opposed to being worked from scratch.

Class Frameworks - Using Subclasses

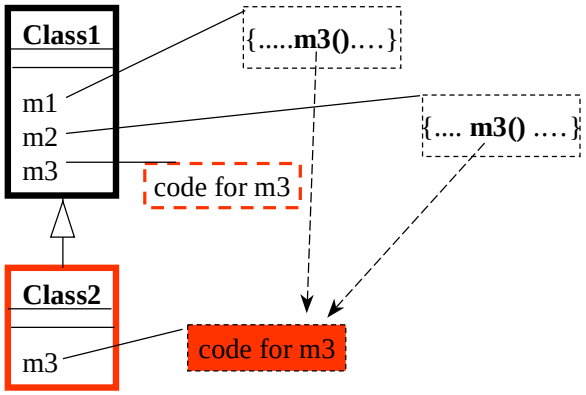
- Subclass inherits superclass implementation
  - Data members always apply
  - Member functions may be filled-in or overridden
  - New data and function members may be defined
- In Java, the subclass also extends the *type* of the superclass
  - Subclass objects presumed substitutable for the superclass type
  - Classes may also be treated as types

The implementation mechanisms provided by subclasses and polymorphism are very powerful. They will take on a somewhat clearer meaning once we have understood the distinction between type and class, and between abstract type-model queries vs. concrete data representations in a class.

Different languages treat classes and types differently.

Java permits interfaces and the interface-extension (subtype) hierarchy to be fully separated from classes. It also allows a class to be used as a type i.e. in the declaration of a parameter, local variable, or data member. Subclasses are treated as subtypes.

Framework-style Design with Subclasses

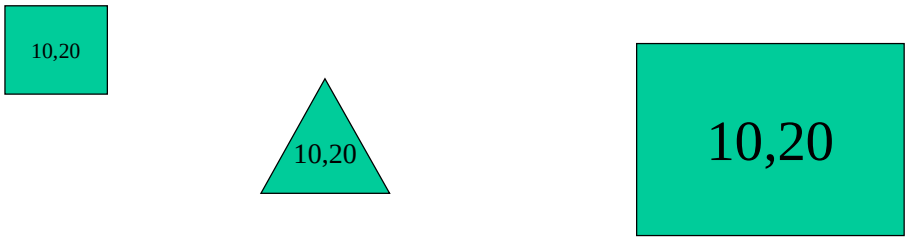


- Plugging-in a small method modifies overall behavior
  - Design of these "plug-points" is important for extensibility
  - Uses the *"template-method"* design pattern

A common style of framework design in OOP uses the *template method* pattern, where a subclass fills in just certain pieces of a method provided by a superclass.

An Example

- **Shape.draw** must have common behavior for all shapes
  - display the shape geometry
  - print coordinates in a font proportional to the bounding box



Here is an example problem.

The "Template Method" pattern

```
class Shape implements IShape {
        private Point center;
        public void draw () {
                BBox b = this.bBox();
                scaleAndPrint (center, b);
                this.render();
        }
        protected abstract BBox bBox();
        protected abstract void render();
}
```

- **Problem**: difficult to document, understand, extend, test
  - Which base methods are affected by an override?
  - To add a new subclass you have to understand base implementation

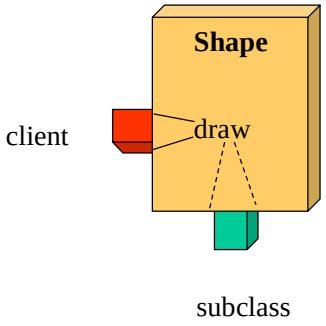
We implement this design by defining a public method *draw* and two protected abstract methods which must be filled in by subclasses of shape: *bBox* and *render*.

This style of design is quite powerful -- we design and build a complete skeleton implementation of behaviors at the superclass, but consciously leave certain placeholders in that implementations at points where subclasses can *plug-in* to adapt the overall behaviors.

However, non-trivial frameworks built in this style are difficult to document, understand, extend, and test. A developer who wishes to extend the framework and looks at the source code does not have an explicit description of which overall methods will be affected by an override in the subclass, or which methods must be simultaneously overridden in some mutually compatible way for the superclass framework to work correctly.

If the developer must understand the framework implementation in order to extend it, then we have an unacceptable degree of coupling between the two, and framework extensions will have to be re-visited and revised anytime the framework undergoes any implementation changes.

Client View vs. Subclass View



- The client does *not* know that *bBox* and *render* are called
  - Client's model and specification of *draw* do not mention them
- Subclass designer must know how *bBox* and *render* are used
  - Simple: expose the entire implementation of *draw* to subclass
  - Better: build abstract model which only exposes essential parts

Note that part of our design is important to the sub-classes, but entirely irrelevant to external clients. Thus, the sub-class needs a wider model of the super-class than does external clients.

Framework implementations commonly require the sub-class designer to understand the entire framework implementation. Catalysis offers an alternate, where the sub-class interface is explicitly described and modeled. This makes the framework extension document as important as the external client description.

Type Models for Subclass Interface

- The subclass uses *boundingBox*, superclass uses *contains*
  - An invariant relates these to each other
    - and hence to all other methods, like bBox() and render()

```
model { BBox boundingBox; }
inv this.contains(p) => boundingBox.contains(p)
```

- Subclasses implement *render* and *bBox* (subclass interface)
  - These must behave in a compatible way

```
model { BBox bBox; }
BBox bBox() spec { return == boundingBox; }
render() spec {  :- nothing written outside of boundingBox; }
```

- Superclass *draw* calls *bBox* and *render*

```
draw spec { -> self.bBox(); ->self.render(); }
```


For example, subclasses must implement *render* and *bBox*. However, though each subclass can implement these in any way it chooses, there is a constraint between these two methods that must be satisfied for the framework to function properly. Specifically, if a subclass *renders* itself outside of the region returned by its *bBox* method, the framework is likely to malfunction.

We have described this with an informal post-condition on the *render* method; this could be formalized if needed.

CBD - Some Missing Pieces!

- Component-based development will take off when:
  - Component interfaces have clear specifications
  - Component libraries include generic designs and architectures
  - Generic components can be customized as needed
  - Composition and refinement of component models is well defined
  - Extraction of abstract patterns is well defined

While it has become very fashionable to talk about component-based development, there are certain enabling factors that need to be present to enable true CBD:

- The interfaces of components must have precise behavioral guarantees

- Design repositories contain implementation as well as design and architectural units

- Components -- design and code -- can be customized and adapted to different needs

- Composition and refinement of generic components is a well-defined operation

- It is possible to abstract out generic fragments from detailed designs

Summary

- We discussed the Catalysis method for CBD using objects
  - Type-modeling and specification
  - Collaborations and composition
  - Refinement of all artifacts
  - Frameworks for recurring patterns across all artifacts
  - From business-level to code with *well-founded use-cases*
- Particularly well suited for component-based development
  - Features very well suited to Java, Java Beans, Active-X, etc.
  - Specification constructs enable large grained components and frameworks
  - Enables precision, assembly, smart tool support

And that, as they say, is that :-)

More details available in the book, and at our web site:

http://www.iconcomp.com

References

- "*Objects, Components, and Frameworks with UML: The Catalysis Approach*", D. D'Souza and A. Wills, http://www.iconcomp.com, Addison Wesley, Winter 1997, ISBN 0-201-31012-0
- *UML 1.0*: http://www.rational.com
- "*Design Patterns: Elements of Reusable Object-Oriented Software*", Erich Gamma, et al, Addison Wesley, 1994.
- "*Analysis Patterns: Reusable Object Models*", Martin Fowler, Addison Wesley, 1997
- *"Contracts: Specifying Behavioural Compositions in Object-Oriented Systems"* Richard Helm et al, OOPSLA/ECOOP 1990
- *"Larch: Languages and Tools for Formal Specification"*, John V. Guttag and James J. Horning (editors), Springer-Verlag Texts and Monographs in CS, 1993

ICON's Worldwide Services

- **Consulting**
  - **Strategic:** Management and technology briefings
  - **Mentoring:** Team and project skills transfer
  - **Development**: analysis, design, architecture
  - **Audits**: designs, architecture, requirements
  - **Process**: Customizing development process

- **Training**
  - **Analysis and Design**: UML, **Catalysis**, Fusion
  - **Implementation**: Java, C++, Smalltalk
  - **Advanced**: Analysis, Design Patterns, Programming...
  - Project management, Technology Overviews

- **HeadStart**
  - Custom fast-track migration
  - Tailored combination package

116

And here are some of the services we provide to projects that are using object and component technology, and our areas of expertise in these technologies.

The services span mentoring, training, consulting, and strategic planning, and cover implementation, architecture, requirements, componentization, and management.

We support methods based on UML, Catalysis, and Team Fusion.

The implementation languages include Java, C++, and Smalltalk.

Contact us if you would like some more information.

Phone: (512) 258-8437 Fax: (512) 258-0086

Email: info@iconcomp.com http://www.iconcomp.com