Software-Konstruktion

Wintersemester 2010/2011

Prof. Dr. Jan Jürjens and Dr.-Ing. Holger Schmidt

TU Dortmund – Department of Computer Science
Software Engineering (LS 14)
http://ls14-www.cs.tu-dortmund.de/

Slides are based on the lecture "Muster- und Komponenten-basierte Softwareentwicklung" by Prof. Dr. Maritta Heisel

Organizational issues I

- Exercise sessions: Holger Schmidt and Gregor Kotainy
- Distribution of lectures and exercises as needed
- Dates
  - Freitags, 14:15-15:00, GB IV - 318
  - Freitags, 14:15-15:00, GB IV - 228
  - Freitags, 15:15-16:00, GB IV - 318
  - Freitags, 15:15-16:00, GB IV - 228
  - Freitags, 16:15-17:00, GB IV - 318
  - Freitags, 17:15-18:00, GB IV - 318
- Course material will be published under

  http://ls14-www.cs.tu-dortmund.de/main2/jj/teaching/index.html

  (check regularly!)

Organizational issues II

Prerequisite: basic knowledge of software engineering as taught in the course "Softwaretechnik"; knowledge of a programming language does not suffice!

Certificate

You have to pass the exam (60 minutes) to get a certificate for this course. The exam schedule will be announced soon on the webpage of this course.

Who studies in another program as the Bachelor?

Content

- Patterns
  - Architectural styles (coarse-grained design)
  - Design patterns (fine-grained design)
  - Idioms (implementation)
- Components
  - Component definition and specification
  - Component models
    - Java Beans
  - Component-based software development process

Software engineering: definition

Software Engineering $\neq$ Programming!

Software Engineering (Balzert):

Goal-oriented provision and systematic use of principles, methods, concepts, notations and tools for team-based development and application of large software systems according to engineering principles. Goal-oriented means e.g. taking costs, time and quality into account.

Software system

A system, whose system components and system elements consist of software.

Software: program + documentation

Phases of software engineering processes

- Analysis
  Goal: understand the problem
- Design
  Goal: obtain structure of software to be built
- Implementation
  Goal: obtain executable software solving the problem
- Testing
  Goal: find defects in implementation

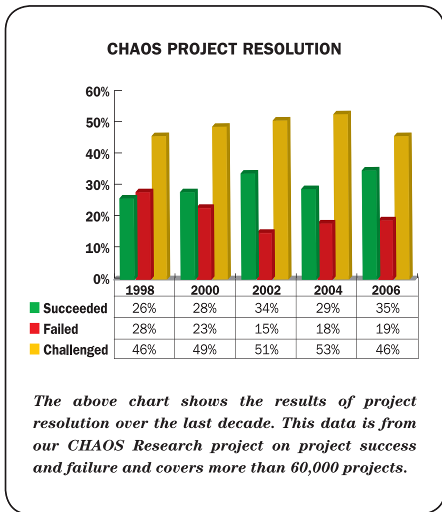
Why do mere programming skills not suffice?

- (Practically) **all** software contains defects.

> *"Software and cathedrals are much the same:*
> *first we build them, then we pray."*
>
> Sam Redwine

- This leads to an immense economic loss and the endangering of human life.
- Why is that so?
- What are new promising areas of research?

Studies of the Standish-Group (CHAOS Research), 1994/1996/1998/2000/2002/2004/2006/2008



	1998	2000	2002	2004	2006
Succeeded	26%	28%	34%	29%	35%
Failed	28%	23%	15%	18%	19%
Challenged	46%	49%	51%	53%	46%

The above chart shows the results of project resolution over the last decade. This data is from our CHAOS Research project on project success and failure and covers more than 60,000 projects.

2008
32%
24%
44%

Why can't software be built in the same way as cars and houses?

Software is something special, because it

- is intangible
- does not wear off through use, but ages because of changes or no changes
- is not restricted through physical laws
- is easily alterable
- is difficult to measure, i.e. describe in a quantitative way
- does not exhibit a continuous but a discrete behavior (no safety margins possible, small causes can have great effects)

Therefore, we need specific methods for constructing software!

Summary

- Computer Science (and thus software engineering) is a very young science.
- Only a small part of software development is programming.
- Due to the special features of software, specific engineering methods are necessary, but have not reached maturity yet.
- An important goal is to develop software with fewer defects.
- For this, promising and exciting new approaches exist.

Recent Developments: From “Art” to “Engineering”

Which steps lead from “Art” to “Engineering”? I

- **Model-based development**
  - Develop sequence of models, each describing different aspects of the software system
  - Models can be analyzed and checked for coherence
- **Object Orientation**
  - Software architecture follows data, not functionality
  - Software as dynamic collection of communicating objects
  - Improved reusability through encapsulation of data
- **Patterns**
  - Templates for the different artifacts generated in software development
  - Useful in all phases of software development
  - Reuse through instantiation

Which steps lead from “Art” to “Engineering”? II

- **Component software**
  - Build software systems from ready-made parts
- **Aspect-oriented programming**
  - Write different programs, each covering different aspects (e.g. computation vs. graphical representation) of the software
  - Compilers combine the different aspect programs to one executable program
- **Software Engineering for special applications**
  e.g. Internet and multimedia applications

Why patterns and components?

- Both belong to the promising new developments
- Both are based on reuse:
  - Patterns allow re-use of **software development knowledge**
  - Components allow re-use of **pre-fabricated software**
- Both can be used in combination

Patterns: basic ideas

- **Templates** for documents set up during software development
- Serve to represent and re-use software development knowledge
- Represent **essence**, abstract from details
- Are used by **instantiation**
- Are available for (almost) all phases of software development

Components: basic ideas

- Assemble software from pre-fabricated parts
- Re-compilation not necessary
- Source code may be inaccessible
- Important: interface descriptions and component models
- Interoperability is an issue

Architectural Patterns

Software architecture

The software we construct for solving a software development problem must be structured further.
That structure is called architecture. It is the result of (coarse-grained) design. It structures the software in terms of

- Components
  These carry out computations. Examples: Filters, data bases, objects, abstract data types
- Connectors
  Means of interaction between components. Examples: procedure calls, pipes, event broadcast

Definition Bass et al. (1998)

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Important points of the definition

- Only properties of components that are externally visible are described.
- These (and only these) constitute the assumptions that can be made by components about one another.
- Internal details that are unimportant for the interaction of components are abstracted from.
- An architecture can define more than one structure. For example, assignment of “modules” to teams, set of parallel processes existing at runtime.

Introduction
Patterns
Architectual Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Why architectures?

All software has an architecture, even if nobody knows it!

However, that architecture should be *explicitly* designed and documented, because this entails that the software

- is better comprehensible
- can be analyzed more easily, e.g. for efficiency
- can be better maintained
- can be implemented systematically

Architectural styles

Introduction
Patterns
Architectual Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

- Are patterns on the level of coarse-grained design, and are also called architectural patterns
- Classify software systems
- The architecture of a software should be an instance of some architectural style

Types of components used in architectures I

Passive components:

- process the requests sequentially and may return values
- used in call-and-return systems

Types of components used in architectures II

Independent components:

- can be implemented as communicating processes
- may initiate actions on their own (active)
- communicate using messages, pipes/streams, or shared memory
- different components can run on one computer with shared memory, or components can be distributed over a network
- exchange data, but do not control each other
- goal: modifiability of the software by decoupling different computations

Types of components used in architectures III

Reasons for independent components:

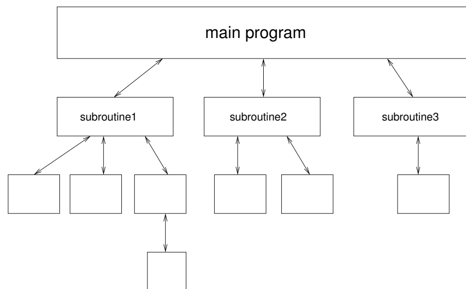
- Software should run on a multiprocessor-platform
- Software could be structured as a set of loosely coupled components, i.e. a component should be able to make reasonable progress while waiting for events from other components
- Performance is important. It can be improved by assigning tasks to the processes and assigning processes to processors.

Structure of software / components I

- function-oriented (FO, main program/subroutine)
- object-oriented (OO)

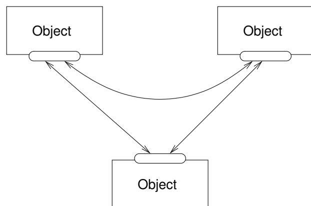
function-oriented software

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References



- Hierarchical decomposition of functionality, based on a *uses*-relation
- One single control-line, directly supported by programming languages
- Implicit subsystem structure: subprograms as modules
- Hierarchical deduction: correctness of a subprogram depends on the correctness of the subprograms it calls.

Object-oriented software



- Orientation of the architecture on the data
- If modifiability and the ability of integration (through well-defined interfaces) is important, consider using an object-oriented design.
- Encapsulation: access only possible through defined operations. The user of a service does not need to know, how it is implemented. The implementation can be changed.
- Disadvantage: object identities must be known.

Interfaces between components / coupling

Types of component coupling:

- call-and-return (for passive components)
- messages / events (for independent components)
- pipes / streams (for all, even network pipes exist)
- shared memory (for all local components)

Call-and-return

Call-and-return (control flow with some associated data):

- Used to connect passive components
- Corresponds to the call of an operation in a programming language
- Technically, a shared memory (stack or processor registers) is used to pass parameters and return values
- Works for function-oriented (main program/subroutine) and object-oriented software
- If the sequence of computations is fixed and components cannot make reasonable progress while waiting for the results of other components, consider using synchronous calls.

Messages / Events I

Messages or Events (control flow with some associated data):

- Asynchronous vs. synchronous (call-and-return) communication
  - between active components usually asynchronous communication is used
  - message queues are used to implement asynchronous communication
  - asynchronous messages cannot have a return value - an additional message in the opposite direction is necessary

Messages / Events II

- Sender-/Receiver relationship
  - 1:1 - The sending component sends a message to one known receiver component
  - 1:1U - The sending component sends a message to a unknown receiver component - The receiver has to register to get the message
  - 1:N - The sending component sends a message to all components that are registered to receive the message (see *Observer* pattern)

Introduction

Patterns

Architectural Patterns

Design Patterns

Idioms

Patterns: Summary

Components

References

Messages / Events III

- Distribution, several components of a software may run:
  - within a single task (asynchronous messages are not necessary)
  - within a single process in separate tasks, but the same memory region
  - locally on one computer in separate processes (communication is between different processes)
  - remote on different computers (communication is over a network connection)

Messages / Events IV

- Data
  - no parameters, only single event (only control flow)
  - only limited data (e.g., a pointer)
  - only simple data types
  - serializable data
  - any object

Messages / Events V

- Implementations:
  - Events in Java (synchronous, 1toN, within single process, any object, OO)
  - Delegates in .NET (asynchronous, 1toN, within single process, any object, OO)
  - Signals and Slots in C++ with QT (asynchronous, 1toN, within single process, any object, OO)
  - Remote procedure calls (Windows RPC/Sun RPC) (synchronous or asynchronous, 1to1, remote, serializable data, FO)
  - Remote Method Invocation (Java RMI) (synchronous, 1to1, remote, serializable data, OO)

Messages / Events VI

- Corba (synchronous/asynchronous since 2000, 1to1, remote, serializable object, OO)
- Windows Events/Mutex/Semaphore received with *WaitForSingleObject* command (asynchronous, 1toN, within single process, no parameters, FO)
- Unix Signals sent with *kill -x* (asynchronous, 1to1, local, no parameters, FO)
- Windows Message Queues (asynchronous, 1to1, remote, serializable data, FO)

Pipes / Streams

Streams (data flow with necessary control messages):

- Network sockets
- Unix pipes (only between components of one computer)
- Windows named pipes (only between components of one computer)

Shared Memory

Shared Memory (can only be used for data flow):

- usually possible within a single process
- if different tasks work on one memory region, synchronization is necessary
- files can be used as a shared memory between different processes
- most operating systems provide functionality to reserve a shared memory that can be used by different processes

Control flow vs. data flow

Introduction
Patterns
Architectual Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

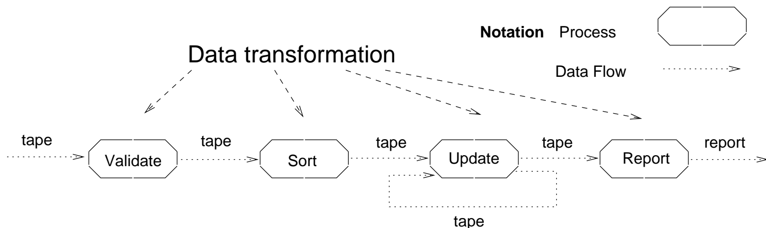
Control flow	Data flow
Decisive question: how does the location of control move through the program?	Decisive question: how does the data move through the program?
Data can go along with control, but is not decisive.	The control is activated where the data is situated.
Important: sequence of computations	Important: availability and transformation of data

Organization of components

- Batch sequential (independent components using file system as shared memory or passive components with call-and-return)
- Pipes & filters (can be implemented with pipes, messages or as call-and-return system)
- Layered architectures (using calls or messages)
- Client-server architecture, using streams (e.g. Sockets) or remote messages (e.g., RPCs)
- Data-centered systems (repositories)
  - Data bases
  - Blackboards
- Event systems (implemented with messages, Observer pattern applied)

Batch sequential

- Processing steps are independent programs that run as different processes
- Each step terminates before the next one begins
- Data are transferred as a whole
- File can be used as a shared memory between the different processes

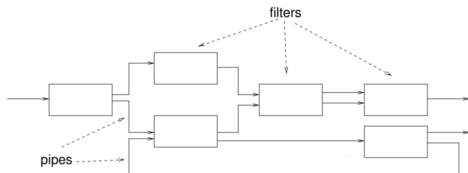


Examples: typical transformational applications such as computing salaries, or the like

Pipes & Filters

Name of that style originates from programs written in the Unix programming environment

- Filters: transform streams of input data into streams of output data in an incremental way
- Pipes: move data from a filter output to a filter input
- General scheme of computation:
  let pipes and filters operate in a non-deterministic manner until no further computations are possible



Specialization: pipelines, i.e., linear sequences of filters

Advantages of pipes & filters architectures

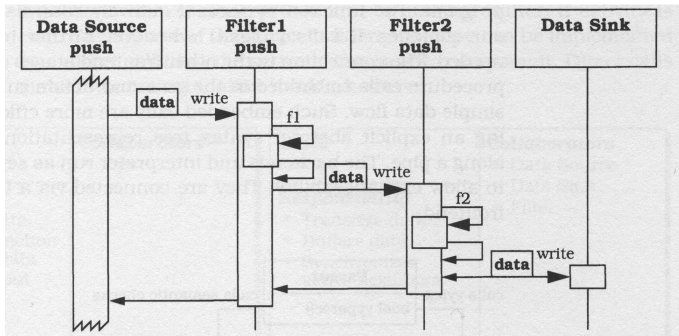
- The overall input-output behavior is determined by a simple composition of the behavior of the individual filters.
- Re-use of filters is possible.
- Easy to maintain and to improve by adding or replacing filters.
- Concurrency is supported in a natural way, because filters can operate independently of each other.
- Can be analyzed well, for example concerning throughput of deadlocks.

Disadvantages of pipes & filters architectures

- Often lead to batch-processing, i.e. concurrency is not utilized
- Not appropriate for interactive applications
- Efficiency may be problematic
- All components have to parse the input

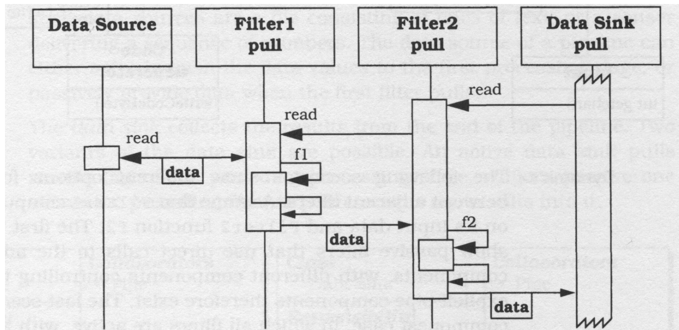
Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) I

Alternative 1: push pipeline with passive filter components and synchronous calls. Activity starts with the data source.



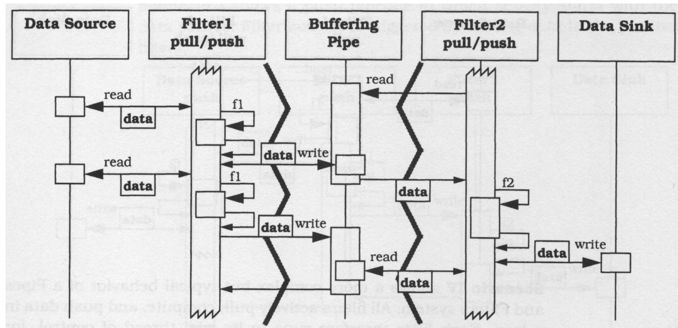
Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) II

Alternative 2: pull pipeline with passive filter components and synchronous calls. Data sink starts the activity by calling for data.



Pipes & filters' dynamic behavior / implementation alternatives (Buschmann et al. (1996)) III

Alternative 3: pipeline with active filter components that pull, process and than push data. Each filter runs in its own thread of control. Buffering pipes are used for communication and synchronize the flow of data.



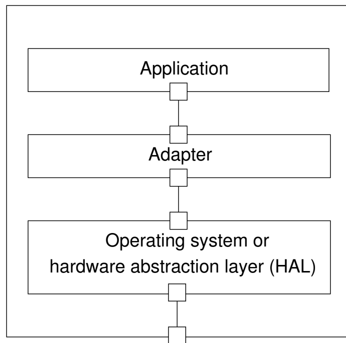
Comparison of batch sequential and pipes & filters

- Both decompose software systems into fixed sequences of computations
- In both cases components interact only through data flow

batch sequential	pipes & filters
coarse-grained, total	fine-grained, incremental
no concurrency	concurrency possible
not interactive	often interactive, but inelegant

Layered architectures

Hierarchical Organization. Each layer offers services for the layers above.



Well-known Example: ISO/OSI-Reference model for communication protocols.

Advantages / disadvantages of layered architectures I

Advantages:

- Design is performed on successive lower abstraction layers, i.e. services are defined at first in an abstract way and then in an increasingly concrete way.
- Can be changed easily, since changes in one layer (should) only effect the adjacent layers.
- Portability is supported.
- Can be implemented as a call-and-return system or composed from independent components.

Advantages / disadvantages of layered architectures II

Disadvantages:

- It is often difficult to identify and clearly separate different abstraction layers.
- The previous reason and reasons of efficiency often lead to *layer bridging* in practice, i.e. not only adjacent layers communicate directly with each other.

Advantages / disadvantages of layered architectures III

Reasons for a layered architecture:

- If the software tasks can be divided into classes, of which one is application-specific and the other is usable for several applications, but platform specific, consider using a layered architecture.
- Also consider using a layered architecture, if the software should be portable or an already developed infrastructure can be used.

Client-server architectures

A server serves several clients, which are usually distributed over a net. Service requests are always initiated by the client, and can be served in a synchronous or asynchronous way.

- Example: web-server and browser (client)
- The repository architecture style is a special client-server architecture
- *Client-dispatcher-server* design pattern is often applied

Data-Centered Systems (Repositories) I

Introduction

Patterns

Architectual Patterns

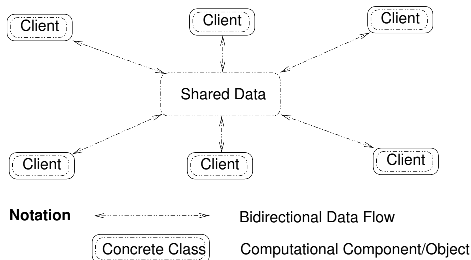
Design Patterns

Idioms

Patterns: Summary

Components

References



Characteristics:

- The integration of data is an important goal.
- The software can be described by describing how the repository can be used and changed by the different parties.
- Components that access the repository are relatively independent from each other, and the repository is independent of them.

Data-Centered Systems (Repositories) II

- New components can be easily added and are not effected by changes of other components

If components act independently from each other, then such a repository architecture is a client-server-architecture at the same time $\Longrightarrow$ Architectural styles are not disjoint!

Databases

The data storage is passive, the sequence of the operations is defined through the input streams.

Blackboards

A blackboard is an active repository: it sends messages to interested components, when certain data has changed. Overlap with event/action-style.

Data-Centered Systems (Repositories) III

Heuristics for Repository Architectures

- Central problem is the storage, representation, administration as well as access to a large number of connected, persistent data.
- Choose a database architecture, if the execution order of the components is determined through a stream of queries and transactions, and if the data are highly structured or in case a commercial database system is available, which can then be used for the desired purpose.
- Choose a blackboard architecture, if consumer and producer of data should be easily exchangeable.
- If it is probable that the representation of data will change, prefer an object-oriented architecture.

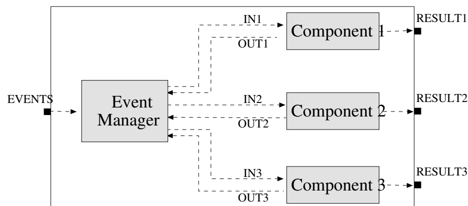
Event systems I

- Also called *Event/Action*-Style
- Independent components do not need to know each other
- Components publish that they offer certain data or services
- Other components announce interest in particular events or data
  $\Longrightarrow$ publish/subscribe-principle
- Often an event- or message manager is responsible for distributing the messages

Choose an event-system, if

- producers and consumers of events should be decoupled.
- scalability is important. Here, new processes can be added, that react to already defined events.

Event systems II



- Each component defines incoming procedure calls and outgoing events in its interface
- The communication among components takes place by publishing events that trigger procedure calls
- Sequence of the called procedures is not deterministic
- Decoupling of implementation and use of components
- Implemented using the *Observer* design pattern

Relation between architectural patterns and design patterns I

Event Systems:

- Implemented using *Observer* pattern

Components:

- Structured using *Facade* pattern
- A component is often implemented using the *Singleton* pattern

User Interface Components:

- Implemented using *MVC* pattern (with *Composite*, *Observer*, *Strategy*, and *Factory Method*)

Relation between architectural patterns and design patterns II

Streams:
If a stream interface is given but messages should be exchanged efficiently, apply

- *Forwarder-Receiver* pattern

Remote Procedure Call (RPC):
In RPC implementations the following design patterns are applied:

- *Client-Dispatcher-Server* pattern to locate the service
- *Proxy* pattern for the operation stubs of the client

What have we learned on architectural patterns?

- The the software system needs to be structured.
- That structure is called architecture of the software system. It consists of components and connectors.
- Software architectures describe the structure of the solution of a problem.
- Software architectures can be classified. These classes are called architectural styles.
- Usually, several architectures can be used to structure a software. These differ in non-functional characteristics (**quality attributes**).

Introduction

Patterns

Architectural Patterns

Design Patterns

Idioms

Patterns: Summary

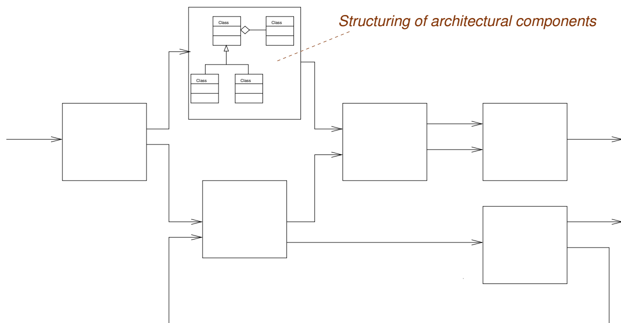
Components

References

Design Patterns

Design patterns (Gamma et al. (1995)): characterized by

- Usage for **detailed design**
- Object-oriented paradigm
- "Description of a family of solutions for a software design problem" (Tichy)



Types of design patterns (Gamma et al. (1995))

- **creational**
  concern the process of object creation
- **structural**
  deal with the composition of classes or objects
- **behavioral**
  characterize the ways in which classes or objects interact and distribute responsibility

Second criterion: **scope**
specifies whether the pattern applies primarily to classes or to objects.

Types of design patterns (Tichy) I

- Coupling/decoupling patterns
  System is divided into units that can be changed independently from each other
  e.g. Iterator, Facade, Proxy
- Unification patterns
  Similarities are extracted and only described at one place.
  e.g. Composite, Abstract Factory
- Data-structure patterns
  Process states of objects independently of their responsibilities
  e.g. Memento, Singleton

Types of design patterns (Tichy) II

- Control flow patterns
  Influence the control flow; provide for the right method to be called at the right time
  e.g. Strategy, Visitor
- Virtual machines
  Receive programs and data as input, execute programs according to data
  e.g. Interpreter
  (Remark: no clear boundary to architectural styles)

Advantages of design patterns (Tichy)

- Improvement of team communication
  Design pattern as "short formula" in discussions
- Compilation of essential concepts, expressed in a concrete form
- Documentation of the "state of the art"
  Help for less experienced designers, not constantly reinventing the wheel
- Improvement of the code quality
  Given structure, code examples

Description of design patterns (Gamma et al. (1995)) I

Name and Classification A good name is important, because it will become part of the design vocabulary.

Intent What does the pattern do? Which problems does it solve?

Also Known As Other familiar names.

Motivation Scenario which illustrates the design problem and how the pattern solves the problem.

Applicability What are the situations in which the design pattern can be applied? How can one recognize these situations?

Structure Class and interaction diagrams.

Participants Classes and objects, which are part of the pattern, as well as their responsibilities.

Description of design patterns (Gamma et al. (1995)) II

Collaborations How do the participants collaborate to carry out their responsibilities?

Consequences What are the trade-offs and results of using the pattern? What aspect of system structure does it let one vary independently?

Implementation What pitfalls, hints, or techniques should one be aware of when implementing the pattern? Are there any language-specific issues?

Sample Code Code fragments in C^{++} or Smalltalk.

Known Uses At least two examples of applications taken from existing systems of different fields.

Related Patterns Similar patterns and patterns that are often used in combination with the described pattern.

Model-View-Controller (MVC)
Gamma et al. (1995); Buschmann et al. (1996)

Introduction

Patterns

Architectural Patterns

Design Patterns

Idioms

Patterns: Summary

Components

References

- Architectural style/design patternhybrid
- Aggregate design pattern out of
  - Composite
  - Observer
  - Strategy
  - Factory Method
- Clear distinction of data (model), data representation on a screen (view) and control of data manipulation or views (controller)

How MVC works – an overview

Introduction

Patterns

Architectual Patterns

Design Patterns

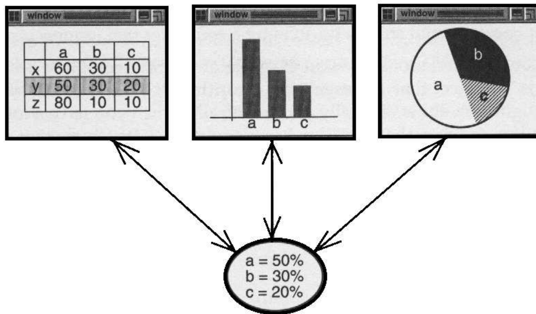
Idioms

Patterns: Summary

Components

References

View(s)



User

Model

Controller

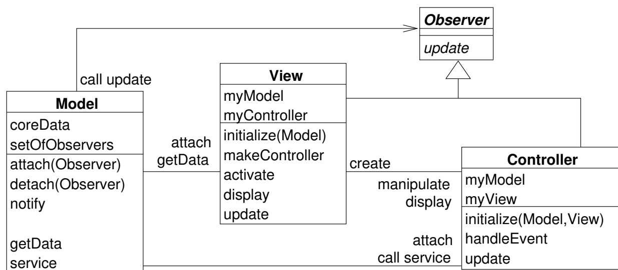
Description of MVC, Trygve M. H. Reenskaug

Intent	Interactive applications with a flexible human-computer interface.
Motivation	Adaptability and reuse
Participants	MVC separates the application into three (independent) components

- *Model* offers core functionality and data
- *View* provides information to the user
- *Controller* handles user input

- All three components are related by a *change-propagation mechanism*.
- *View* and *Controller* constitute the user interface.

MVC class diagram



- **Model**: does not know View or Controller beforehand, announces change by calling *update*, related components request model state by *getData*, if needed keep data in a data base
- **View**: connected to Model, displays data (visually, acoustically, or similar) normally on a screen
- **Controller**: administrates Views, manipulates data on behalf of the user, "brain" of the application

Design Patterns of MVC in detail

Composite I

Classification object/structural

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets you treat individual objects and compositions of objects uniformly.

Also Known As —.

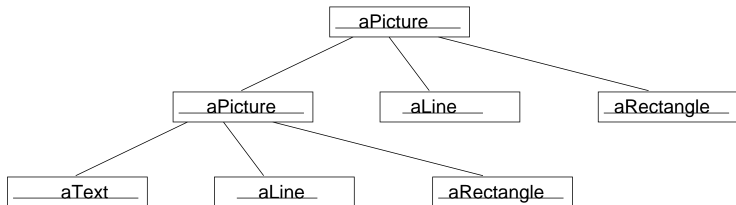
Motivation

Users can build complex diagrams out of simple components by using graphics applications.

Problem: Code that uses the corresponding classes must treat primitive and container objects differently, even if most of the time the user treats them identically. The Composite pattern describes how a recursive composition can be designed so that the client does not have to distinguish between primitive objects and containers.

Composite II

Example:



common operations: *draw*(), *move*(), *delete*(), *scale*()

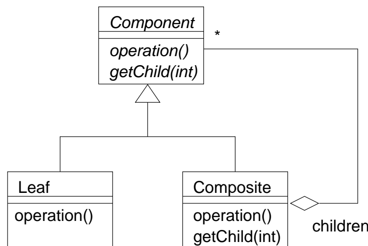
Composite III

Applicability

Use the Composite pattern when

- you want to represent part-whole hierarchies of objects.
- you want clients be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Structure (abstract classes and operations are noted in *italics*)



Composite IV

Participants

- *Component* (graphic)
  - declares the interface for objects in the composition
  - implements default behavior for the interface common to all classes, as appropriate
  - declares an interface for accessing and managing its child components
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate
- *Leaf* (rectangle, line, text etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition

Composite V

- *Composite* (picture)
  - defines behavior for components having children
  - stores child components
  - implements child-related operations in the *Component* interface
- *Client* (not contained in the class diagram)
  - manipulates objects in the composition through *Component* interface

Collaborations

Clients use the *Component* class interface to interact with objects in the composite structure. If the recipient is a leaf, then the request is handled directly. If the recipient is a composite, then it usually forwards the request to its child components, possibly performing additional operations before and/or after forwarding.

Composite VI

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects.
  Whenever client code expects a primitive object, then it can also take a composite object.
- makes the client simple
  Clients can treat composite structures and individual objects uniformly. Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite object. This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

Composite VII

- makes it easier to add new kinds of components. Newly defined subclasses of *Composite* or *Leaf* work automatically with existing structures and client code. Clients don't have to be changed for new component classes.
- can make your design overly general
  The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Composite VIII

Implementation Gamma et al. (1995) considers the following aspects:

1. Explicit parents references
   Should be defined in the *Component* class.
2. Sharing components
   Can be useful to reduce storage requirements, but destroys tree structure.
3. Maximizing the *Component* interface
   Necessary to make clients unaware of the specific *Leaf* or *Composite* classes they are using. Default implementation in *Component* can be overwritten in subclasses.

Composite IX

4. Declaring the child management operations
   Declaration of *add*- and *remove*-operations in the class *Component* results in transparency; all components can be treated uniformly. Costs safety, because meaningless operations can be called, e.g. adding to objects to leafs. Defining child management in the *Composites* class gives safety, but is at the expense of transparency (leaves and composites have different interfaces).
5. Should *Component* implement a list of components? Incurs a space penalty for every leaf.
6. Child ordering
   When child ordering is an issue, applying the Iterator pattern is recommended.

Composite X

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

7. Caching to improve performance
   Useful, if the compositions have to be traversed or searched frequently.
8. Who should delete components?
   In languages without garbage collection, it's usually best to make a composite responsible for deleting its children when it's destroyed.
9. What's the best data structure for storing components?
   Depends on aspects of efficiency.

Composite XI

Sample Code

Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies.

```
1  class Equipment {
2       public:
3           virtual Equipment();
4           const char* Name() { return _name; }
5           virtual Watt Power();
6           virtual Currency NetPrice();
7           virtual Currency DiscountPrice();
8           virtual void Add(Equipment*);
9           virtual void Remove(Equipment*);
10          virtual Iterator<Equipment*>* CreateIterator();
11     protected:
12         Equipment(const char*);
13     private:
14         const char* _name;
15     };
```

Composite XII

Equipment declares operations that return the attributes of a piece of equipment, like its power consumption and cost. A *CreateIterator*-operation returns an iterator for accessing its parts.

Further classes such as *FloppyDisk* as class for leaves and *CompositeEquipment* for composite equipment are defined in the Gamma et al. (1995).

Composite XIII

Known Uses

- *View*-class in Model/View/Controller
- Composite structure for parse trees
- Portfolio containing assets

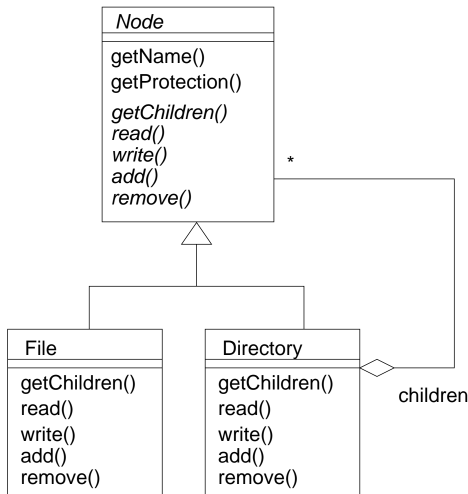
Related Patterns

- Often the component-parent link is used for a **Chain of Responsibility**.
- **Decorator** is often used with composites. When decorators and composites are used together, they will usually have a common parent class.
- **Flyweight** lets you share components, but they can no longer refer to their parents.
- **Iterator** can be used to traverse composites.
- **Visitor** localizes operations and behavior that would otherwise be distributed across *Composite* and *Leaf* classes.

Example: file system

- File system should be able to handle file structures of any size and complexity.
- *Directories* and (basic) *files* should be distinguished.
- The code, e.g. for selecting the name of a directory should be the same as for files. The same holds for size, access rights, etc.
- It should be easy to add new types of files (e.g. symbolic links).

Application of the composite pattern



Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Observer (or: Publisher/subscriber) I

Classification object/behavioral

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As Dependents, Publish-Subscribe

Observer (or: Publisher/subscriber) II

Applicability

Use the Observer pattern when

- change to one object requires changing others, and you do not know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who these objects are.
- data changes a one place, but many other components depend on this data
- the number and identity of dependent components is not known a priori or may change over timer
- polling is not feasible.

Observer (or: Publisher/subscriber) III

Structure

Subject
Attach(Observer)
Detach(Observer)
Notify()

observers

Observer
Update()

ConcreteSubject
subjectState
GetState()
SetState()

subject

ConcreteObserver
observerState
Update()

Observer (or: Publisher/subscriber) IV

Participants

- *Subject*
  - knows its observers
  - provides an interface for attaching and detaching Observer objects
- *Observer*
  - defines an update interface for objects that should be notified of changes in a subject
- *ConcreteSubject*
  - stores state of interest to *ConcreteObserver* object
  - sends a notification to its observers when its state changes
  - Also called **publisher**.

Observer (or: Publisher/subscriber) V

- *ConcreteObserver*
  - maintains a reference to a *ConcreteSubject* object
  - stores state that should stay consistent with the *ConcreteSubject*'s
  - implements the `update`-interface of *Observer*
  - components/objects depend on changes
  - Also called **subscriber**.

Observer (or: Publisher/subscriber) VI

Dynamics:

aConcreteSubject
aConcreteObserver
anotherConcreteObserver
setState()
notify()
update()
getState()
update()
getState()

Related Patterns Mediator, Singleton

Example: file system

- When the name of a file or directory is changed (`setName`), the representation of the name on the display will be updated, too.

Application of the Observer pattern

Strategy I

Classification object/behavioral

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

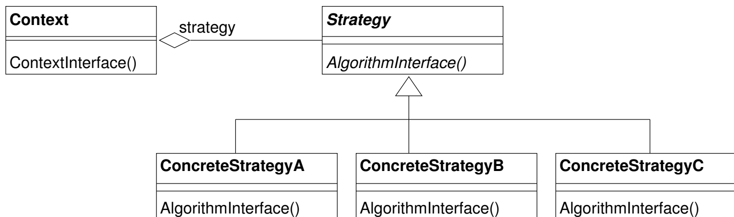
Also Known as Policy

Applicability Use the Strategy pattern when

- many related classes differ only in their behavior. Strategy provides a way to configure a class with one of many behaviors.
- you need different variants of an algorithm.
- an algorithm uses data that clients should not know about.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

Strategy II

Structure



Strategy III

Participants

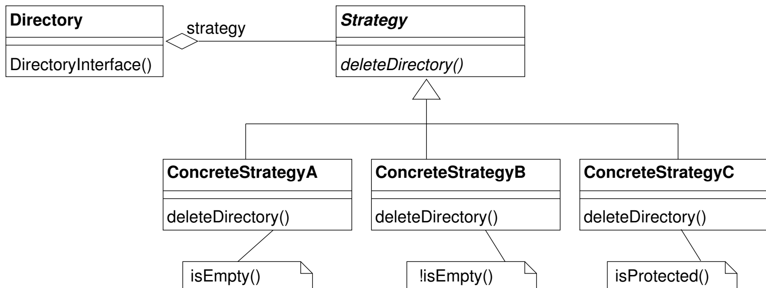
- *Strategy*
  - declares an interface common to all supported algorithms. Context uses this interface to call algorithm defined by a *ConcreteStrategy*.
- *ConcreteStrategy*
  - implements the algorithm using the *Strategy* interface.
- *Context*
  - is configured with a *ConcreteStrategy* object
  - may define an interface that lets *Strategy* access its data

Related Patterns Flyweight

Example: file system

- It is not allowed to delete directories which contain files or are write protected.

Application of the Strategy pattern



Factory Method I

Classification creational

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Also Known As Virtual Constructor

Applicability Use the Factory Method pattern when

- a class cannot anticipate the class of objects it must create.
- a class wants its subclass to specify the object it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Factory Method II

Structure

Product

Creator

FactoryMethod()
AnOperation()

ConcreteProduct

<<create>>

ConcreteCreator

FactoryMethod()
AnOperation()

Factory Method III

Participants

- *Product* (Files)
  - defines the interface of objects the *FactoryMethod*() creates
- *ConcreteProduct* (Text-File)
  - implements the *Product* interface
- *Creator* (Application)
  - declares the *FactoryMethod*(), which returns an object of type *Product*
- *ConcreteCreator* (Open Office)
  - overrides the *FactoryMethod*() to return an instance of a *ConcreteProduct*

Related Patterns

Abstract Factory, Template Method, Prototypes

Introduction

Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary

Components

References

Example: file system

- The file system offers the creation of files, where the kind of file to be created (.txt, .ods, .xls) depends on the particular application.

Application of the Factory Method pattern

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Singleton I

Classification object/creational

Intent Ensure a class only has one instance and provide a global point of access to it.

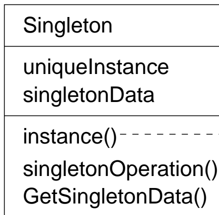
Motivation
It's important for some classes to have exactly one instance (e.g., printer spooler). That class should be responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance.

Applicability Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Singleton II

Structure



if uniqueInstance = null
then uniqueInstance := new Singleton
endif
return uniqueInstance

Participants

- *Singleton*
  - defines an *instance* operation that lets clients access its unique instance.
  - may be responsible for creating its own unique instance.

Singleton III

Collaborations

- Clients access a singleton instance solely through the singleton's *instance*-operation.

Consequences

- Controlled access to the sole instance.
- Improvement over global variables.
- Permits refinement of operations and representation through subclassing
- Permits a variable (!) number of instances.

Related Patterns Abstract Factory, Builder, Prototype

Example: file system

Consider users in a multi-user system:

- User logs in to the system.
  - generates an object of the class *UserSession*
- We want to ensure that
  - a only a maximum number of user sessions exist per user.
  - user sessions are only generated if authentication was successful.
- Basic concept:
  - singleton-pattern
  - variation necessary

Applying the Singleton pattern

- Declare class *UserSession* to be a singleton.
- Instantiation of *instance* is named *createUserSession*.
- Extend implementation of *createUserSession* by further case distinctions (number of user sessions is smaller than allowed maximum, successful authentication).

Facade I

SWK

JJ+HS

Introduction

Patterns

Architectual Patterns

Design Patterns

Idioms

Patterns: Summary

Components

References

Classification object/structural

Intent Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. A facade can provide a simple default view of the subsystem that is good enough for most clients.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade II

Introduction

Patterns

Architectural Patterns

Design Patterns

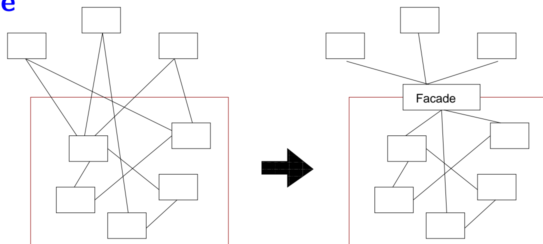
Idioms

Patterns: Summary

Components

References

Structure



Participants

- *Facade*
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects
- subsystem classes
  - implement subsystem functionality
  - handle work assigned by the facade object
  - have no knowledge of the facade, i.e. no reference to it

Facade III

Consequences The facade

- shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- promotes weak coupling between subsystems and clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
- doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Related Patterns Abstract Factory, Mediator

Example: file system

Introduction

Patterns
- Architectual Patterns
- Design Patterns
- Idioms
- Patterns: Summary

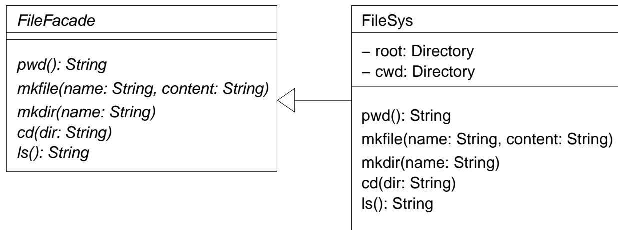
Components

References

Uniform interface for file system:

- File system API contains different classes, whose interaction is difficult to understand.
- In particular, the admissible consequences for generating file structures are not clear.
- In real file systems: uniform interfaces for handling the different phenomena file, directory, alias, ...

Applying the Facade pattern



- Implementing *FileSys* of *FileFacade* contains two private attributes
- Creation routine generates a *root*-directory "/" and sets *cwd* to *root*
- *pwd* calls *cwd.getName*
- *mkfile* calls *cwd.add*
- ...

Proxy I

Classification object/structural

Intent Provide a surrogate or placeholder for another object to control access to it.

Also Known As Surrogate

Applicability Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

Some situations in which the Proxy pattern is applicable:

1. A *remote proxy* provides a local representative for an object in a different address space.
2. A *virtual proxy* creates expensive objects on demand (delayed loading, delayed generation).

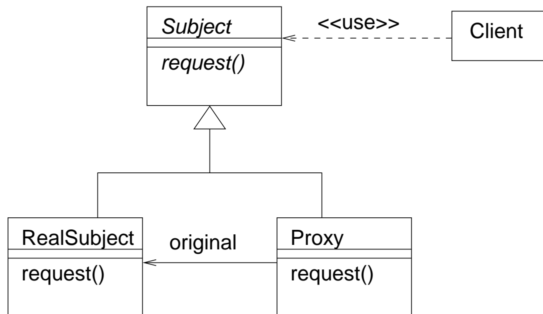
Proxy II

3. A *protection proxy* controls access to the original object. Protection proxies are useful when objects should have different access rights.
4. A *smart reference* is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number or references to the real object so that it can be freed automatically when there are no more references (also called *smart pointer*)
   - loading a persistent object into memory when it's first referenced
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy III

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Structure



Proxy IV

Participants

- *Proxy*
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to *Subject*'s so that a proxy can be substituted for the real subject
  - controls access to the real subject and may be responsible for creating and deleting it
- *Subject*
  - defines common interfaces for *RealSubject* and *Proxy*, so that the proxy can be used anywhere a real subject is expected
- *RealSubject*
  - defines the real object that the proxy represents

Related Patterns Adapter, Decorator

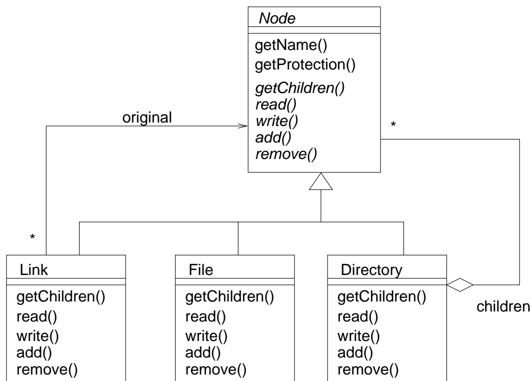
Example: file system

Introduction of aliases

- file alias
  - “symbolic link” in Unix
  - “alias” in MacOS
  - “shortcut” in Windows95+
- operations on files and aliases
  - alias permits all operations that are possible on originals
  - forwards operations to the original
  - special interpretations of operations is possible in special cases (e.g., for copying)

Applying the Proxy pattern to the file system

New class *Link* as proxy for *Node*



Client-Dispatcher-Server I

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Classification structural

Intent/Problem Buschmann et al. (1996)

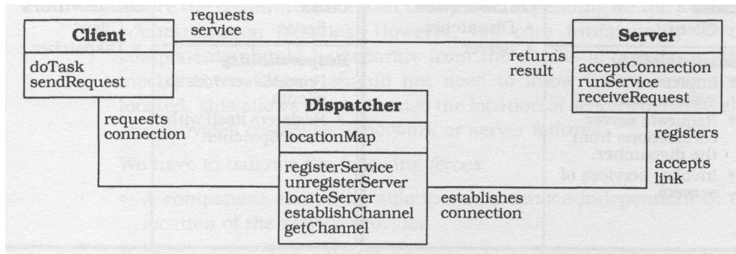
- Software system uses servers distributed over a network
- Connection between components have to be established before communication
- Core functionality should be separated from communication details
- Clients should not need to know where servers are located

Client-Dispatcher-Server II

Also Known As -

Motivation/Applicability Services are located on different servers

Structure



Client-Dispatcher-Server III

Participants/Consequences/Implementation

- Provide a **dispatcher** to act as an intermediate layer between client and server
- Dispatcher implements a name service to provide location transparency
- Dispatcher establishes the communication
- **Servers** provide services to other components
- Servers have unique names and are connected to the dispatcher
- **Clients** rely on the dispatcher to locate a particular service and to establish a connection

Client-Dispatcher-Server IV

Dynamics

Client

Dispatcher

Server

registerService

doTask

getChannel

locateServer

establishChannel

acceptConnection

sendRequest

Channel

receiveRequest

runService

possible process boundary

possible process boundary

Client-Dispatcher-Server V

Sample Code see Buschmann et al. (1996)

Known Uses RPCs, CORBA

Related Acceptor and Connector

Forwarder-Receiver (Peer-to-peer) I

Classification structural

Intent/Problem Buschmann et al. (1996)

- Commonly distributed applications use efficient low-level mechanisms for inter-process communication (e.g., TCP/IP, message queues)
- Low-level mechanisms often introduce dependencies on the underlying operating system and network protocol, which restricts portability
- Higher-level mechanisms like remote procedure calls are less efficient
- Communication mechanism should be exchangeable
- The senders should only need to know the names of their receivers
- The communication should not have major impact on performance

Forwarder-Receiver (Peer-to-peer) II

Also Known As Peer-to-peer

Motivation/Applicability Efficient communication between peers

Structure

Forwarder
marshal
deliver
sendMsg

Receiver
receive
unmarshal
receiveMsg

IPC

sendMsg

receiveMsg

Peer 1
service

1:1

optional process boundary

1:1

Peer 2
service

receiveMsg

sendMsg

Receiver
receive
unmarshal
receiveMsg

IPC

Forwarder
marshal
deliver
sendMsg

Forwarder-Receiver (Peer-to-peer) III

Participants/Consequences/Implementation

- Distributed **peers** collaborate to solve a particular problem.
- A peer may act as a client, a server, or both.
- The details of the underlying communication mechanism are hidden from peers
- System-specific functionality (name mapping to physical locations, communication channel establishment, marshaling) is encapsulated into separate components.
- A **forwarder** marshals the data and sends messages to other peers
- A **receiver** receives and unmarshals the data.

Forwarder-Receiver (Peer-to-peer) IV

Dynamics

Peer P1
Forwarder Forw1
Receiver Recv1
Receiver Recv2
Forwarder Forw2
Peer P2
service
sendMsg
marshal
optional process boundary
deliver
IPC msg
receiveMsg
receive
unmarshal
sendMsg
marshal
deliver
IPC msg
receiveMsg
receive
unmarshal

What have we learned on design patterns? I

- Design patterns are object-oriented patterns at detailed design level.
- They are closer to implementation than architectural styles.
- According to the classification of Gamma et al. (1995), there are behavioral, creational and structural patterns.
- Design patterns support achieving desirable properties in implementing object-oriented software, e.g. independent modification of parts, limitation of communication paths etc.

What have we learned on design patterns? II

- We have presented and used the following patterns for MVC:
  1. Composite
  2. Observer
  3. Strategy
  4. Factory Method

  plus the patterns

  5. Singleton
  6. Facade
  7. Proxy
  8. Client-dispatcher-server
  9. Forwarder-Receiver

Introduction

Patterns
- Architectural Patterns
- Design Patterns
- Idioms
- Patterns: Summary

Components

References

Idioms

Characteristics

- Specific patterns for (object-oriented) programming languages
- Low abstraction level
- Describe, how certain aspects of components or relations between components can be implemented by means of a specific programming language

Literature

- Buschmann et al. (1996)
- Coplien (1992)
- Coplien (1998)
  `http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html`

Application

- Solution of implementation-specific problems in a certain programming language, e.g.
  - memory management
  - creation of objects
- Implementation of design patterns
- Description of programming styles, e.g.
  - names for operations
  - formatting of source code
- Simplified communication between developers

Idiom for implementing “Singleton” in C++ I

Introduction
Patterns
Architectural Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Name Singleton (C++)

Problem An implementation of the Singleton design pattern is needed to ensure that only one instance of a class exists at runtime.

Solution Change the constructor of the corresponding class to a private operation. Declare a static attribute `theInstance`, which refers to the single instance of the class. Initialize the pointer in the class declaration with null. Define a public static operation `getInstance()`, which returns the value of the attribute. When the operation is called for the very first time, the single instance of the class is constructed using the operator `new`. Furthermore, this instance is assigned to the attribute `theInstance`.

Idiom for implementing “Singleton” in C++ II

Introduction
Patterns
Architectual Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Example

```
class Singleton {
   static Singleton *theInstance;
   Singleton();
   public:
      static Singleton *getInstance() {
         if (! theInstance)
            theInstance = new Singleton;
         return theInstance;
      }
};
//...
Singleton* Singleton::theInstance = 0;
```

Idiom for Implementing “Singleton” in Smalltalk I

Name Singleton (Smalltalk)

Problem An implementation of the Singleton design pattern is needed to ensure that only one instance of a class exists at runtime.

Solution Override the operator `new` of the corresponding class such that it triggers an exception. Add the class attribute `TheInstance` to the class, which contains the single instance of the class. Implement the operation `getInstance()`, which returns this instance. When the operation is called for the very first time, the single instance of the class is constructed using the operator `super new`. Furthermore, this instance is assigned to the attribute `TheInstance`.

Idiom for Implementing “Singleton” in Smalltalk II

Introduction
Patterns
Architectual Patterns
Design Patterns
Idioms
Patterns: Summary
Components
References

Example

```
new
   self error: 'cannot create new object'

getInstance
   TheInstance isNil ifTrue:
      [TheInstance := super new].
   ^TheInstance
```

Example of an Idiom in C++: Counted Pointer I

Example Problem of the C++ memory management. Several clients have a reference to a commonly used object. This issue leads to two unwanted situations:

1. A client object deletes the commonly used object while it is referenced by another client.
2. No client object references the commonly used object, but the object was not deleted.

Context Memory management of dynamically allocated, multiple-referenced instances of a class.

Example of an Idiom in C++: Counted Pointer II

Problem Objects will be passed as parameters to functions using pointers. The following *forces* rule:

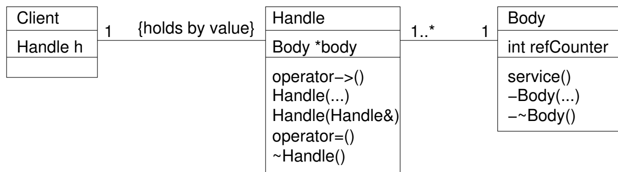
- several clients refer to the same object
- "dangling references" should be avoided
- object that are not referenced should be deleted
- solution should contain only a small portion of additional client code

Example of an Idiom in C++: Counted Pointer III

Solution

- counting of references of multiple-referenced objects
- *body class* will be extended by reference counter
- only a *handle class* is allowed to refer to objects of the body class
- objects will be passed as value parameters and hence automatically allocated and deleted
- handle class manages reference counter of body class instances
- by overloading the operator “->” in `object->operation()` using `operator->()` in the handle class, its instances can be used as if they were pointers on body class instances

Example of an Idiom in C++: Counted Pointer IV



Implementation

1. Declare the constructors and the destructor of the body class as private or protected methods to prevent uncontrolled creation and deletion of objects.
2. Declare the handle class as a friend class of the body class; hence it can access the features of the body class.
3. Extend the body class by a reference counter (`refCounter`).

Example of an Idiom in C++: Counted Pointer V

4. Add an attribute to the handle class pointing at a body object.
5. Implement the copy constructor (`Handle(Handle&)`) and the assignment operator of the handle class by copying the pointer to the body object and incrementing the reference counter. Implement the destructor (`~Handle`) of the handle class by decrementing the reference counter and deleting the body class object (if the reference counter reaches 0).
6. Implement the public arrow operator of the handle class as follows:

```
Body* operator->() const { return body; }
```

7. Extend the handle class by one or more constructors, which create a body class instance the handle object points at. Each of these constructors initializes the reference counter of its body class object with 1.

Example of an Idiom in C++: Counted Pointer VI

Sample solution C++-Code ...

Variants *CountedBody*-Idiom (cf. Coplien 1992): each client has the illusion that it uses its own body class object, even though it is referenced by other clients. The body class object must be copied if a client modifies it.

Introduction

Patterns
- Architectural Patterns
- Design Patterns
- Idioms
- Patterns: Summary

Components

References

What have we learned?

- Idioms are patterns on a low level of abstraction.
- They are tailor-made for specific (object-oriented) programming languages.
- They constitute concrete guidelines to solve specific programming problems in a specific programming language.

Summary

Patterns for different software development phases

- Architectural styles
  Structuring the software using components and connectors
- Design patterns
  Fine-grained design of architectural components, communication between components or objects
- Idioms
  Realization of a problem solution using a specific programming language

Conclusions

- There are patterns for practically all phases of software development.
- Patterns enable developers to construct software systematically.
- Patterns have the potential to improve not only the software development process, but also the resulting software products.

Components

Component technology - introduction

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

- New trend in software technology
- Basic idea: build software system from smaller (already developed and tested) parts
- Re-build (compiling) of components usually not nessecary; we distingish between
  - White-box components (source code available), and
  - Black-box components (only binary available)
- Interface desciptions and component model/standard are important
- Current Technologies: (Enterprise) Java Beans, OSGi Service Platform, Component Object Model (COM), Corba Component Model (CCM)
- The component approach tries to apply standard engineering methods to software development

Definitions of "Component" I

- Generally, "component" only means "part of ..."
- Doug McIlroy coined the term "software-component" at the Garmisch conference in October 1968
- The term is overloaded, e.g., for software architectures

Definitions of "Component" II

Some definitions for (black-box) components:

- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. (Szyperski et al. (2002))
- A package of software that is independently developed and that defines interfaces for the services it provides and the services it requires. (D'Souza and Wills (1998))
- A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard (Heineman and Councill (2001)).
- More definitions, see Szyperski et al. (2002), Chapter 11.

Component forms I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Features of components have in the different life-cycle states (Cheesman and Daniels (2001)):

- To use a component it must conform to the **Component Standard** in use, like Enterprise Java Beans (EJB) or Microsoft COM+.
- **Component Specification**: valid definition of the component.
- **Component Interface** or just **Interface** is a major part of the component specification.
- It should be possible to replace one **Component Implementation** with another with the same Component Specification.
- **Installed Component**: installed copy of the implementation.
- **Component Object**: instance of an Installed Component.

Component forms II

Component Specification

*

supportedInterface 1..*

Component Interface

1

* realization

Component Implementation

1

* installation

Installed Component

1

* instance

Component Object

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

Design by Contract

What are preconditions and postconditions good for?

Contracts in daily life, Meyer (1997)

- Contractual partners are clients and sellers or service providers.
- Both expect advantages from the contract and are willing to make a commitment.

Example

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

I want to travel from Berlin to Duisburg.

	Commitments	**Advantages**
Passenger	Pay ticket Be there at departure time **must keep precondition**	getting to Duisburg **Has advantages from the postcondition**
Traffic provider	Must take the passenger to Duisburg **Must guarantee postcondition**	receives the price for the ticket; does not have to take passengers who have not paid or did not arrive in time **Can assume precondition**

Advantages of explicit contracts

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Meyer:

A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope.

Application to software

A contract is a formal agreement between a software / a class and its environment / clients. It specifies the rights and duties for both sides.

Contents of implementation contracts

Precise description of the functional properties of instances of a class at its **interface**:

- What does the class require from its clients?
- What does the class guarantee to its clients?
- What combinations of attribute values are permitted?

Contract

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

If the client fulfills the requirements of the server, then the server will provide the specified functionality.

- The client can rely on the assertions of the server. The internals of the server class are of no interest to the client.
- If the client does not fulfill the requirements of the server, then the server has no obligations whatsoever, it can behave arbitrarily (including breakdown).
- **It is not the server that has to test if the precondition holds, but the client!**

Example: Stack (generic class)

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

class	Stack[T]
attribute	nb_elements: integer
	max_size: integer
method	empty(): Boolean
	full(): Boolean
	push(x: T)
	pop()
	top(): T
end class	Stack[T]

Specification of the stack operations with preconditions and postconditions I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- empty()

  pre true
  post **noChange and**
  Result = true ⇔ nb_elements = 0

- full()

  pre true
  post **noChange and**
  Result = true ⇔ nb_elements = max_size

- push(x: T)

  pre **not** full
  post **not** empty **and**
  nb_elements = **old** nb_elements + 1 **and**
  top = x

Specification of the stack operations with preconditions and postconditions II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- pop()

  pre **not** empty
  post **not** full **and**
  nb_elements = **old** nb_elements - 1
  and "top element of the stack is deleted"

- top(): T

  pre **not** empty
  post **noChange and**
  Result = "top element of the stack"

SWK

JJ+HS

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Commitments and advantages

	Commitments	Advantages
Client	Call *push*(x) only if stack is not full **Must keep precondition**	Element x is put on stack, top() results in x, *nb_elements* increases by 1. **Has advantages from postcondition**
Server	Makes sure that x is placed on the stack **Must guarantee postcondition**	Unnecessary to handle the case if stack is full. **Can assume precondition**

Method specification – precondition

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Methods form the operational interface between client and server.
Hence, a contract on the level of methods must describe the condition under which a client is allowed to call a method (precondition) and the effect the server guarantees in that case (postcondition).

Precondition: Predicate on the parameters of the method and the attributes of the class.

Requirement of the server to its clients – must hold when method is called.
Example: **not** full

Method specification – postcondition

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

The effect of a method describes the state that holds after the method has terminated and the values of the output parameters in terms of the input parameters and the state that holds when the method is called.

Postcondition: Relation between input parameters, attributes of the class **before** executing the method, and the attributes of the class **after** executing the method, and the output parameters.

Example: **not** empty **and** nb_elements = **old** nb_elements + 1 **and** top = x

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Class invariant

Not all combinations of attribute values describe an admissible instance of a class.

class invariant: Property describing an integrity condition on the attributes of a class.

Example: $0 \leq$ nb_elements $\leq$ max_size **and** max_size ≥ 1

The class invariant is implicitly contained in the pre- and postconditions of all methods!

Contract – Relation between client and server

Commitment of the client

- Satisfy preconditions of creation routines (constructor)
- Satisfy preconditions of methods

Commitment of the server:

- Creation routines establish class invariant
- Methods keep class invariant
- Methods establish postconditions

Relation to abstract data types

- Classes correspond to implementations of abstract data types (ADTs).
- In an ADT specification of a stack, we would have the following axioms:
  pop(push(x,s)) = s
  top(push(x,s)) = x
- These axioms cannot be expressed in terms of pre- and postconditions of single methods, because they express relations between several different methods.
- However, a stack implementation should guarantee that the axioms are fulfilled.

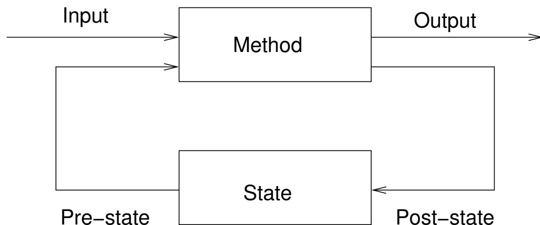
Contracts and inheritance

If an inheritance hierarchy is part of an interface, i.e., clients can access servers polymorphically, then a subclass must keep all contracts of all superclasses.

- The class invariant must imply all the class invariants of the superclasses.
- Preconditions of re-defined methods must be implied by the preconditions of the super-methods.
- Postconditions of re-defined methods must imply the postconditions of the super-methods.

These conditions guarantee that a client does not experience any “surprises” when using a polymorphic server without knowing its exact dynamic type.

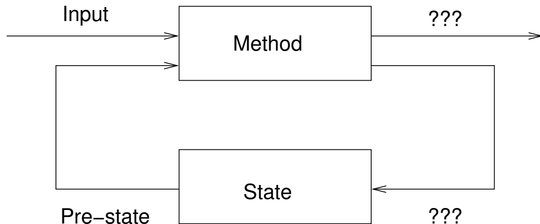
Design by Contract: Overview



Precondition describes input and pre-state

Postcondition describes relation between input/pre-state and output/post-state

Design by Contract: If precondition is not satisfied



If we call a method with the precondition not satisfied

- we do not know if there is any output and – if so – how it looks like
- we do not know if the method will terminate and – if so – how the post-state will look like

Advantages of Design by Contract

- Contracts make given restrictions explicit.
- Clear distribution of functionality at the interface between client and server.
- Avoiding unnecessary checks through overly defensive programming.
- Abstraction from the implementation of the server (replaceability).
- (Partial) checks at runtime by **assertions**.

What have we learned?

- The principle of design by contract makes explicit the obligations of users and providers of services.
- The caller of a method/a procedure (i.e., the client) must guarantee that the precondition is fulfilled; the server must in turn guarantee that the postcondition is fulfilled.
- Assertions should be added to the code and checked at runtime. Thus, errors are easier to find.

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

Components and OO

This part describes approaches for structuring object-oriented software systems into (white-box) components. Often, these components cannot be built separately.

Components and OO I

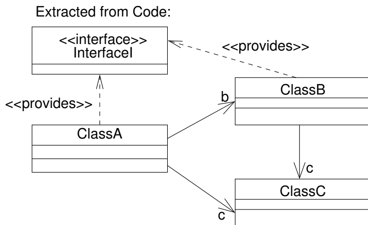
Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

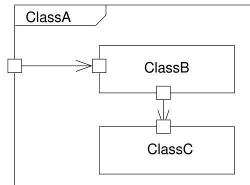
In the source code and in UML models, we usually have associations between classes. These associations can be either references to other components, or the referenced objects are part of one component.

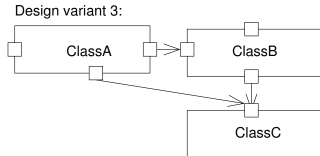
```
 class ClassA implements InterfaceI{
   private ClassB b;
   private ClassC c;
}
class ClassB implements InterfaceI{
   private ClassC c;
}
class ClassC {
    ...
}
```

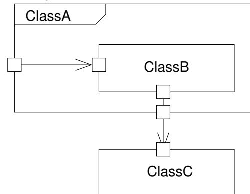
Components and OO II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References









Components and OO III

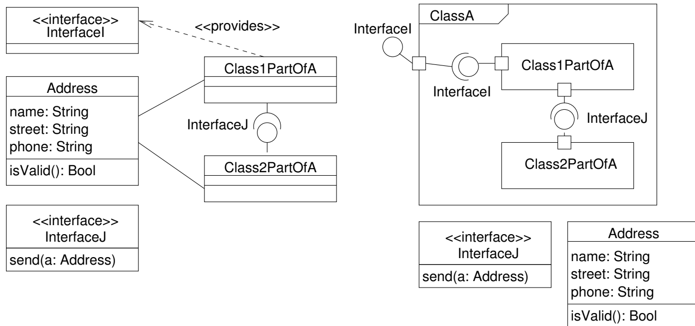
Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Be careful

Not all objects can be clearly associated to a certain component: some objects are used to exchange complex data between components and exchanged as parameters, e.g. a user object is created in the user interface component and sent to the application component for further processing.

Components and OO IV

Example:



Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component coupling levels I

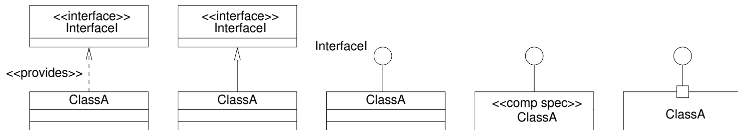
A component may be:

- Just an object of a class
  - May use other objects to provide its functionality
  - The public operations of the class represent the component interface
  - It is not clear which of the objects used to provide the functionality (associated) are part of that component, and which are not.
  - Other objects created by this object can be considered to be part of the component
  - Usually are not built separately

Component coupling levels II

- Object with explicit provided interfaces
  - May use other object to provide its functionality
  - Still not clear which of the objects used to provide the functionality are part of that component
  - Implementation can be better replaced
  - Usually are not built separately

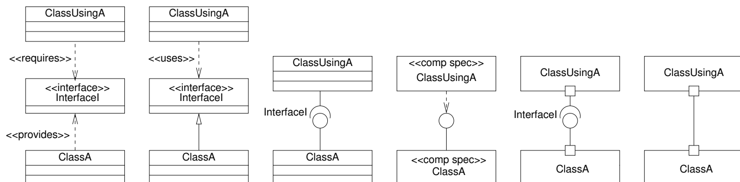
Notation: Class with provided Interface / Lollipop notation / Component according Cheesman and Daniels (2001):



Component coupling levels III

- Object with explicit provided and required interfaces
  - Loose coupling, other components used to provide the functionality are connected during instantiation or initialization
  - Advantage: components can be easily tested separately
  - Other object used to provide the functionality may be created, and they are considered to be part of the component

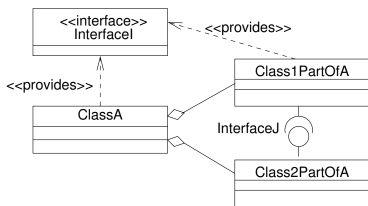
Notation for 2 connected classes / Lollipop notation / Component according Cheesman and Daniels (2001) / Composite Structure:

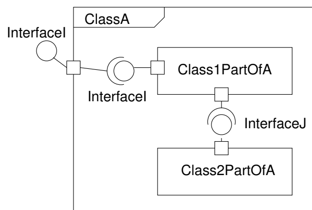


Component coupling levels IV

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Composition in class diagrams and composite structure diagrams:





Component coupling levels V

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Object with explicit provided and required interfaces that makes use of a component standard (e.g., providing events or messages) to communicate with other components
  - Loose coupling, other components used to provide the functionality are connected at run-time
  - Advantage: components can be easily tested separately
  - Usually, can be built separately

Same notation as for objects with explicit provided and required interfaces

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component coupling levels VI

- All components are separate processes that communicate using events or messages
  - Loose coupling, other components used to provide the functionality are connected at run-time
  - Advantage: components can be easily tested separately
  - Usually, can be built separately

Same notation as for objects with explicit provided and required interfaces

Simple Java Components

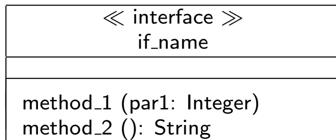
This part describes an implementation approach for components with explicit provided and required interfaces.

Implementation of components with explicit provided and required interfaces

- A component has provided and required interfaces that can be connected with other components.
- A component only uses functionality from its required interfaces, from the programming language, and a limited set of operations of the operating system (e.g., tasks, threads, memory allocation, timers, messages, synchronization mechanisms).
- Provided and required interfaces are represented by interface classes.
- Interface operations are called synchronously.
- Advantage: These classes / components can be easily tested separately.

Implementation of interfaces in Java

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

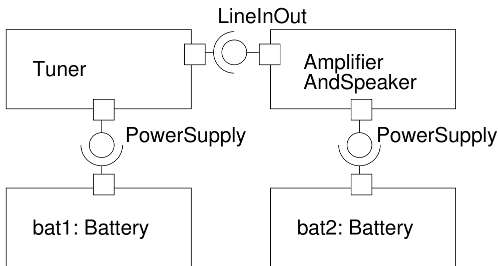



```
package project_name;
public interface if_name {
    public void method_1 (int par1);
    public String method_2 ();
}
```

The project name should be added as a package. Otherwise additional parameters are necessary to compile the project. Note: `int` is a simple data type, and `String` is a class.

Implementation of provided interfaces in Java I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References



Each provided interface is defined as an interface class, e.g.:

```
public interface LineInOut {
   public void transmitMusic();
}

public interface PowerSupply {
   public void powerOn();
}
```

Implementation of provided interfaces in Java II

A component can implement / provide several interfaces, e.g.:

```
public class AmplifierAndSpeaker implements
                           LineInOut, PowerSupply  {
   public AmplifierAndSpeaker (){} //constructor

   public void transmitMusic() { Play;}
   public void powerOn() { Action2;}
}
```

All provided operations must be implemented as methods.

Implementation of required interfaces in Java I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

A component can use / require several interfaces, defined as interface classes.

```
public class Tuner implements PowerSupply {
   private LineInOut outputDevice;
   public Tuner(){ outputDevice = NULL; }
   public void connectTo(LineInOut par) {outputDevice = par;}

   public void powerOn() {
     while (true) {
        if (outputDevice!=NULL) outputDevice.transmitMusic();
   }
}
```

Implementation of required interfaces in Java II

- The required interfaces become private attributes (`outputDevice` of type `LineInOut`).
- The component has to provide methods to connect the component to the required components (`connectTo`). In these connect methods, the private attributes are initialized.
- Via these private attributes, the connected components can be used. They should only be used if they are initialized (`if (outputDevice!=NULL) ...`).

Alternatively, it is possible to leave out the method `connectTo` and initialize the connected interface in the constructor.

- The component `Tuner` also provides the interface `PowerSupply` and implements the method `powerOn`.

Implementation of required interfaces in Java III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

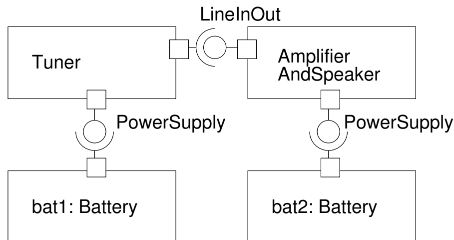
```
public class Battery {
   private PowerSupply suppliedDevice;

   public Battery(){ suppliedDevice=NULL }

   public void connectTo(PowerSupply suppliedDev) {
        suppliedDevice = suppliedDev;
        suppliedDevice.powerOn();
   }
}
```

The component `Battery` powers on the supplied device when connected. It requires the interface `PowerSupply`.

Implementation of required interfaces in Java IV

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

The components *bat1*, *bat2*, *myTuner*, and *myAmp* can be connected as follows:




```
AmplifierAndSpeaker myAmp = new AmplifierAndSpeaker();
Tuner myTuner = new Tuner();
Battery bat1 = new Battery();
Battery bat2 = new Battery()
myTuner.connectTo(myAmp);
bat1.connectTo(myTuner);
bat2.connectTo(myAmp);
```

Component specifications

Structural notations for components:

- Composite structure diagrams
- Class diagrams
- Component diagrams

Additionally to the structure, the behavior of the components must be described using

- Pre- and postconditions for all interface operations (design by contract)
- Sequence diagrams
- State machines

What have we learned?

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- In object orientation, a component can take different forms. These different forms come along with different coupling levels.
- Advantage of loosely coupled components: they can be built and tested separately.
- Provided and required interfaces of components implemented in Java are represented by interface classes. The component has to provide methods to connect the component to the required component.

JavaBeans

JavaBeans I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

JavaBeans

- are **reusable software components** for Java.
- can be manipulated visually in a builder tool (e.g., Sun's NetBeans).
- are classes written in the **Java** programming language.
- **encapsulate many objects** into a single object (the bean).
- conform to the following **convention**: JavaBeans are
  - **serializable**.
  - have a **no-argument constructor**.
  - allow access to **properties** using getter and setter methods.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

JavaBeans II

More information:

- Sun's JavaBeans product webpage: `http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp`
- Sun's JavaBeans API webpage: `http://java.sun.com/javase/technologies/desktop/javabeans/api/index.html`
- Sun's JavaBeans tutorials: `http://java.sun.com/docs/books/tutorial/javabeans/`

Events

Events

- are a mechanism for propagating **state change notifications** between a *source* JavaBean and one or more *target* JavaBeans.
- are the basis to plug JavaBeans together in an **application builder**.
- can be **caught and processed** by JavaBeans.
- have many different uses, but a common example is their use in a window system toolkit for delivering notifications of mouse actions, widget updates, keyboard actions, etc.

Event Model I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- **Event notifications** are propagated from sources to listeners by Java method invocations on the target listener objects.
- Each distinct kind of event notification is defined as a distinct Java method. These methods are then grouped in `EventListener` interfaces that inherit from `java.util.EventListener`.
- Event listener classes identify themselves as interested in a particular set of events by implementing some set of `EventListener` interfaces.

Event Model II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- The state associated with an event notification is normally encapsulated in an event state object that inherits from `java.util.EventObject` and which is passed as the sole argument to the event method.
- Event sources identify themselves as sourcing particular events by defining registration methods and accept references to instances of particular `EventListener` interfaces.
- In circumstances where listeners cannot directly implement a particular interface, or when some additional behavior is required, an instance of a custom **adaptor class** may be interposed between a source and one or more listeners in order to establish the relationship or to augment behavior.

Properties

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Properties

- are **attributes** of a Java Bean that can affect its appearance or its behavior.
- Example: a GUI button might have a property named “Label” that represents the text displayed in the button.
- can be accessed by other JavaBeans calling their **getter and setter methods**.
- typically are **persistent**, so that their state will be stored away as part of the persistent state of the JavaBean.
- can have arbitrary types, including both built-in Java types such as `int` and class or interfaces types such as `java.awt.Color`.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Accessor Methods

- For **readable properties** there will be a getter method to read the property value.
- For **writable properties** there will be a setter method to allow the property value to be updated.
- For simple properties the accessor type signatures are:

  simple setter `void setFoo(PropertyType value);`
  simple getter `PropertyType getFoo();`

Indexed Properties I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- An **indexed property** supports a range of values. Whenever the property is read or written one specifies an index to identify which value is required.
- Property indexes must be of type `int`.
- For indexed properties the accessor type signatures are:

indexed setter

```
void setter(int index, PropertyType value);
```

indexed getter

```
PropertyType getter(int index);
```

Indexed Properties II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

array setter

```
void setter(PropertyType values[]);
```

array getter

```
PropertyType[] getter();
```

- The indexed getter and setter methods may throw a `java.lang.ArrayIndexOutOfBoundsException` runtime exception if an index is used that is outside the current array bounds.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Bound Properties I

- Sometimes when a JavaBean property changes then either the JavaBeans container (i.e., a program that uses the JavaBean) or some other JavaBean may wish to be notified of the change.
- A JavaBean can choose to provide a change notification service for some or all of its properties.
- Such properties are commonly known as **bound properties**, as they allow other components to bind special behavior to property changes.
- The `PropertyChangeListener` event listener interface is used to report updates to simple bound properties.

Bound Properties II

- If a JavaBean supports bound properties then it should support a pair of event listener registration methods for `PropertyChangeListener`:

add listener

```
public void
addPropertyChangeListener
(PropertyChangeListener x);
```

remove listener

```
public void
removePropertyChangeListener
(PropertyChangeListener x);
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Bound Properties III

- When a property change occurs on a bound property the JavaBean should call the `PropertyChangeListener.propertyChange` method on all registered listeners, passing a `PropertyChangeEvent` object that encapsulates the name of the property and its old and new values.
- The event source should fire the event after updating its internal state.

Example: JavaBean Person I

The class Person has a property name, that can be changed using setName(). After a change, the JavaBean informs all listeners of this change.

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;

public class Person
{
  private String name = "";
  private PropertyChangeSupport changes =
    new PropertyChangeSupport(this);

  public void setName(String name)
  {
    String oldName = this.name;
    this.name = name;
    changes.firePropertyChange("name", oldName, name);
  }
```

Example: JavaBean Person II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
  public String getName()
  {
    return name;
  }

  public void addPropertyChangeListener(
    PropertyChangeListener pcl)
  {
    changes.addPropertyChangeListener(pcl);
  }

  public void removePropertyChangeListener(
    PropertyChangeListener pcl)
  {
    changes.removePropertyChangeListener(pcl);
  }
}
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Example: Reaction to Property Change I

Registered `PropertyChangeListener` can react to the `PropertyChangeEvent`.

```
public class ReportChange implements PropertyChangeListener {
  @Override
  public void propertyChange(PropertyChangeEvent e)
  {
    System.out.printf("Property '%s': '%s' -> '%s'%n",
      e.getPropertyName(), e.getOldValue(), e.getNewValue());
  }
}
```

Example: Reaction to Property Change II


```
Person person = new Person();
ReportChange reportChange = new ReportChange();

person.addPropertyChangeListener(reportChange);

person.setName("Ulli");
// expected output: Property 'name': '' -> 'Ulli'
person.setName("Ulli");
// no output expected
person.setName("Chris");
// expected output: Property 'name': 'Ulli' -> 'Chris'
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Constrained Properties I

- Sometimes when a property change occurs some other bean may wish to validate the change and reject it if it is inappropriate.
- We refer to properties that undergo this kind of checking as **constrained properties**.
- In Java Beans, constrained property setter methods are required to support the `PropertyVetoException`. This documents to the users of the constrained property that attempted updates may be vetoed.

Constrained Properties II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

The following operations in the setter method for the constrained property must be implemented in this order:

1. Save the old value in case the change is vetoed.
2. Notify listeners of the new proposed value, allowing them to veto the change.
3. If no listener vetoes the change (no exception is thrown), set the property to the new value.

Constrained Properties III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- A simple constrained property might look like:

```
PropertyType getFoo();
void setFoo(PropertyType value)
  throws PropertyVetoException;
```

- In the body of a setter method, the `fireVetoableChange` method is invoked on the `VetoableChangeSupport` attribute of the Java Bean before the `firePropertyChange` method is invoked on the property that is changed.

Constrained Properties IV

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- A simple setter method for a constrained property might look like:

```
public void setFoo(boolean foo)
  throws PropertyVetoException{
    boolean oldValue = this.foo;
    vetos.fireVetoableChange("foo",
      oldValue, foo);
    this.foo = foo;
    changes.firePropertyChange("foo",
      oldValue, foo);
}
```

Constrained Properties V

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- The `VetoableChangeListener` event listener interface is used to report updates to constrained properties. If a bean supports constrained properties then it should support a pair of event listener registration methods for `VetoableChangeListeners`:

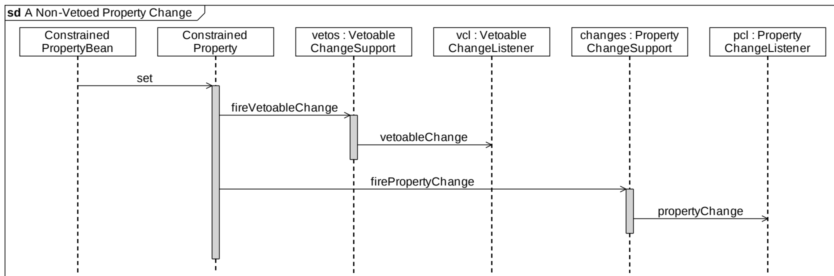
```
public void addVetoableChangeListener
  (VetoableChangeListener x);
public void removeVetoableChangeListener
(VetoableChangeListener x);
```

Constrained Properties VI

- When a property change occurs on a constrained property the bean should call the `VetoableChangeListener.vetoableChange` method on all registered listeners, passing a `PropertyChangeEvent` object that encapsulates the name of the property and its old and new values.

Constrained Properties VII

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirement Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References



Constrained Properties VIII

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

- If the event recipient does not wish the requested edit to be performed it may throw a `PropertyVetoException`. It is the source bean's responsibility to catch this exception, revert to the old value, and issue a new `VetoableChangeListener.vetoableChange` event to report the reversion.
- The initial `VetoableChangeListener.vetoableChange` event may have been relayed to a number of recipients before one vetoes the new value.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Constrained Properties IX

- If one of the recipients vetoes, then one has to make sure that all the other recipients are informed (fire another `VetoableChangeListener.vetoableChange` event) that the old value is restored. The source may choose to ignore vetoes when reverting to the old value.
- The event source should fire the event before updating its internal state.

Example: Reaction to Property Change I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Registered PropertyChangeListener can react to the PropertyChangeEvent.

```
public class ReportChangeVeto implements VetoableChangeListener {
  @Override
  public void vetoableChange(PropertyChangeEvent e)
    throws PropertyVetoException
  {
    if ( "Name".equals( e.getPropertyName() ) )
      if ( "Ulli".equal.( e.getNewValue()) )
        throw new PropertyVetoException( "Not with me", e );
  }
}
```

Example: Reaction to Property Change II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
Person person = new Person();
ReportChangeVeto reportChangeVeto = new ReportChangeVeto();

person.addVetoableChangeListener(reportChangeVeto);

try
{
  person.setName("Ulli");
}
catch ( PropertyVetoException e )
{
  // expected output: java.beans.
  //    PropertyVetoException: Not with me
  e.printStackTrace();
}
```

Example: Reaction to Property Change III

sd A Vetoed Property Change

Constrained PropertyBean
Constrained Property
VetoableChange Support
vcl: VetoableChange Listener
ovcl: VetoableChange Listener

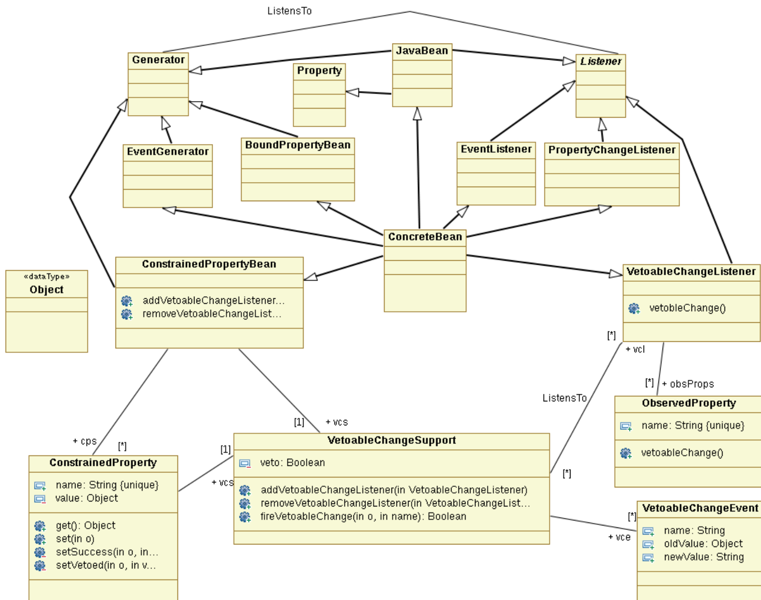
set
fireVetoableChange
vetoableChange
vetoableChange
throw (PropertyVetoExceptio n)
vetoableChange
vetoableChange
throw (PropertyVetoExceptio n)

JavaBeans Component Model

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

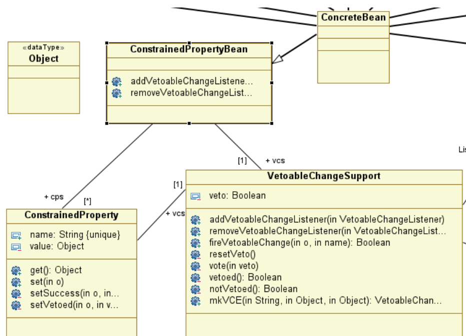
- **Component model** to specify the characteristics of a JavaBean according to this model.
- Based on a formalization of Sun's JavaBeans model by Heisel et al. (2002).
- Described as a metamodel using a UML class diagram and OCL constraints.
- Instances of this metamodel constitute concrete JavaBean specifications.

JavaBeans Metamodel I
Class Model



JavaBeans Metamodel II

OCL Constraints

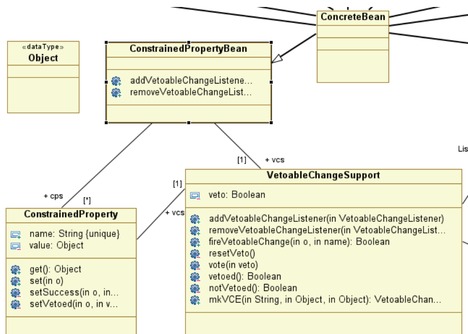


The same `vcs` object must be used by all objects belonging to the set cps.

```
context ConstrainedPropertyBean
inv:  self.cps->forAll(
  cp:ConstrainedProperty|cp.vcs=self.vcs)
```

JavaBeans Metamodel III

OCL Constraints



The content of `value` is returned.

```
context ConstrainedProperty.get
post:  result=self.value
```

Creating a Simple JavaBean I

Write the `SimpleBean` code. Put it in a file named `SimpleBean.java`.

```
import java.awt.Color;
import java.beans.XMLDecoder;
import javax.swing.JLabel;
import java.io.Serializable;

public class SimpleBean extends JLabel
 implements Serializable {
   public SimpleBean() {
      setText( "Hello world!" );
      setOpaque( true );
      setBackground( Color.RED );
      setForeground( Color.YELLOW );
```

Creating a Simple JavaBean II


```
        setVerticalAlignment( CENTER );
        setHorizontalAlignment( CENTER );
    }
}
```

- `SimpleBean` extends the `javax.swing.JLabel` graphic component and inherits its properties, which makes the `SimpleBean` a visual component.
- `SimpleBean` also implements the `java.io.Serializable interface`.

Compiling the JavaBean and Generating a Java Archive (JAR) File I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Create a manifest, the JAR file, and the class file `SimpleBean.class`.
- Use the Apache Ant (`http://ant.apache.org/`) tool to create these files.
- Apache Ant is a Java-based build tool that enables one to generate XML-based configurations files as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project default="build">

 <dirname property="basedir" file="${ant.file}"/>

 <property name="beanname" value="SimpleBean"/>
 <property name="jarfile"
  value="${basedir}/${beanname}.jar"/>
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Compiling the JavaBean and Generating a Java Archive (JAR) File II

```
<target name="build" depends="compile">
    <jar destfile="${jarfile}"
     basedir="${basedir}" includes="*.class">
 <manifest>
     <section name="${beanname}.class">
 <attribute name="Java-Bean" value="true"/>
     </section>
 </manifest>
    </jar>
</target>
```

Compiling the JavaBean and Generating a Java Archive (JAR) File III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
<target name="compile">
    <javac destdir="${basedir}">
 <src location="${basedir}"/>
    </javac>
</target>

<target name="clean">
    <delete file="${jarfile}">
 <fileset dir="${basedir}" includes="*.class"/>
    </delete>
</target>
</project>
```

Loading the JavaBean into the GUI builder of the NetBeans IDE I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- It is recommended to save an XML script in the `build.xml` file, because Ant recognizes this file name automatically.
- Load the JAR file. Use the NetBeans IDE GUI Builder to load the jar file.

1. Start NetBeans.
2. From the file menu select "New Project" to create a new application for the bean. You can use "Open Project" to add the bean to an existing application.
3. Create a new application using the "New Project Wizard".

Loading the JavaBean into the GUI builder of the NetBeans IDE II

4. Select a newly created project in the list of projects, expand the “Source Packages” node, and select the “Default Package” element.
5. Click the right mouse button and select “New - JFrameForm” from the pop-up menu.
6. Select the newly created form node in the project tree. A blank form opens in the GUI builder view of an editor tab.
7. Open the palette manager for “Swing/AWT components” by selecting “Palette Manager” in the “Tools” menu.
8. In the “Palette Manager” window select the beans components in the palette tree and press the “Add from JAR” button.

Loading the JavaBean into the GUI builder of the NetBeans IDE III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

9. Specify a location for the `SimpleBean` JAR file and follow the “Add from JAR Wizard” instructions.
10. Select the palette and properties options from the “Windows” menu.
11. Expand the beans group in the palette window. The `SimpleBean` object appears. Drag the `SimpleBean` object to the GUI builder panel.

Loading the JavaBean into the GUI builder of the NetBeans IDE IV

The following figure represents the `SimpleBean` object loaded in the GUI builder panel:

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Inspecting the JavaBean's Properties and Events

- The `SimpleBean` properties will appear in the Properties window.
- For example, one can change a background property by selecting another color.
- To preview the form, use the "Preview Design" button of the GUI builder toolbar.
- To inspect events associated with the `SimpleBean` object, switch to the events tab of the "Properties" window.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

What have we learned?

- JavaBeans are (mainly) visual components according to a component model by Sun.
- Consequently, they can be used to build graphical user interfaces using builder tools such as NetBeans.
- Formalization of the JavaBeans component model using UML class diagram, sequence diagrams, and OCL.
- Construction of a simple JavaBean.

OSGi Service Platform

OSGi Service Platform I

Wütherich et al. (2008)

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- OSGi defines a **dynamic component model** for Java, ie. components can be installed, updated and uninstalled without stopping or restarting the platform.

- Components provided by OSGi are called **bundles**.
  A bundle
  - contains an additional file with descriptive information, e.g. about provided and required interfaces.
  - can implement a **service**. Services are registered at a central **Service Registry** where other bundles can request it.
  - can be in different states (e.g. installed, active). The bundle lifecycle can be managed by the **OSGi Framework API**.

OSGi Service Platform II

Wütherich et al. (2008)

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

OSGi

- used to stand for **O**pen **S**ervice **G**ateway **i**nitiative.
- is a **standard** defined by the OSGi Alliance (`http://www.osgi.org`).
- is used in applications ranging from mobile phones to the Eclipse IDE[1].
- is realized by open source (e.g. Eclipse Equinox) and commercial implementations.
- consists of two parts: **OSGi Framework** and **OSGi Standard Services**

[1] **I**ntegrated **D**evelopment **E**nvironment

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

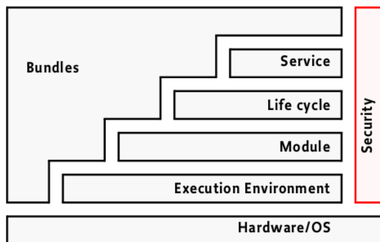
References

OSGi Framework

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

OSGi Framework (OSGi Alliance (2010b)) I

- The OSGi Framework implements a container for bundles.
- The functionality of the framework is divided into the following layers:



Execution Environment Defines the Java environment that is needed to execute the OSGi Framework.

Module Defines a component model for Java.

Lifecycle Defines the states of a bundle.

Service Defines a service model.

Security Defines security relevant aspects.

OSGi Framework (OSGi Alliance (2010b)) II

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

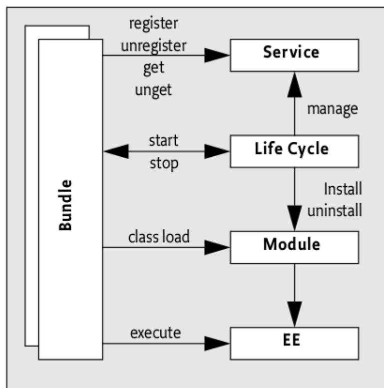
Component Interaction

Component Specification

Provisioning and Assembly

References

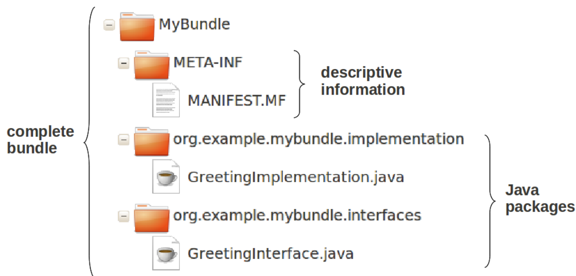
Interactions between layers:



Bundles

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- A **Bundle**
  - represents a component in the OSGi Framework.
  - consists of one or more Java packages.
  - is deployed as a **J**ava **AR**chive (JAR) with additional descriptive information.



- The descriptive information is stored within the **bundle manifest** `MANIFEST.MF`.

Example: Bundle Manifest


```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: My first bundle
Bundle-SymbolicName: org.example.mybundle
Bundle-Version: 1.0.0
```

Bundle Manifest Header	Optional	Description
`Bundle-ManifestVersion`	yes	Number corresponds to version of the OSGi specification (2 for current version).
`Bundle-Name`	yes	Defines a readable name for the bundle.
`Bundle-SymbolicName`	no	Bundle symbolic name and version must identify a unique bundle.
`Bundle-Version`	yes	Specifies the version of the bundle (default value is `0.0.0`).

Export and Import of Packages I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- By default, the classes contained in a bundle are **not** visible to classes from other bundles.
- In order to use classes of one bundle in another bundle, they must be **exported** and **imported**.
- In OSGi, only packages (and thereby the contained classes) may be exported and imported.

Export and Import of Packages II

SWK

JJ+HS

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- In order to offer a provided interface, a bundle must export the package containing the interface. Therefore the following line has to be added to the corresponding `MANIFEST.MF`:

```
Export-Package: org.example.mypackage,
                org.example.anotherpackage
```

- A bundle that requires these interfaces has to import the packages. This is done by adding the following line to the `MANIFEST.MF` of that bundle:

```
Import-Package: org.example.mypackage,
                org.example.anotherpackage
```

- The OSGi Framework resolves these dependencies by matching the imports and exports automatically as soon as both bundles are installed.

Export and Import of Packages III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- An exported package can be supplied with a version:

```
Export-Package:
  org.example.mypackage;version="1.0.0"
```

  (The default value is `0.0.0`.)

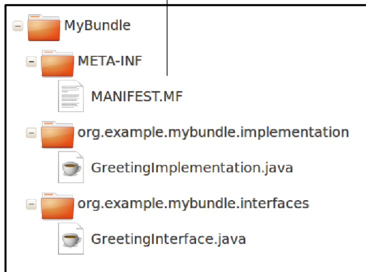
- For an imported package, a version range can be specified:

```
Import-Package:
  org.example.mypackage;version="[1.1.0,1.5.0)"
```

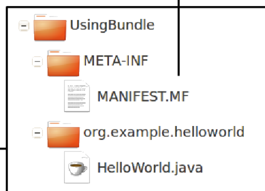
  (i.e. `org.example.mypackage` can only be imported if its version number is greater than or equal to `1.1.0` and less than `1.5.0`)

Example: Bundles with provided and required interfaces I

`Export-Package: org.example.mybundle.interfaces`



`Import-Package: org.example.mybundle.interfaces`



Example: Bundles with provided and required interfaces II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Bundle "MyBundle"

Package containing the interface:

```
package org.example.mybundle.interfaces;
public interface GreetingInterface {
  public void sayHello();
}
```

Package containing the implementation:

```
package org.example.mybundle.implementation;
import org.example.mybundle.interfaces.GreetingInterface;
public class GreetingImplementation implements
  GreetingInterface {
  public void sayHello() {
    System.out.println("Hello!");
  }
}
```

Example: Bundles with provided and required interfaces III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

MANIFEST.MF of bundle "MyBundle":

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with provided interface
Bundle-SymbolicName: org.example.mybundle
Bundle-Version: 1.0.0
Export-Package: org.example.mybundle.interfaces;
                version="1.0.0"
```

Example: Bundles with provided and required interfaces IV

Bundle "UsingBundle"

Package using the interface:

```
package org.example.helloworld;
import org.example.mybundle.interfaces.GreetingInterface;
public class HelloWorld {
  public HelloWorld(GreetingInterface gi) {
    gi.sayHello();
  }
}
```

Example: Bundles with provided and required interfaces V

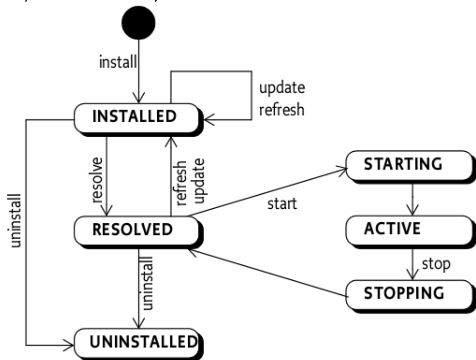
Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

MANIFEST.MF of bundle "UsingBundle"

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with required interface
Bundle-SymbolicName: org.example.usingbundle
Bundle-Version: 1.0.0
Import-Package: org.example.mybundle.interfaces;
                version="[1.0.0,1.5.0)"
```

Bundle Lifecycle

- A bundle that is installed within the OSGi Framework can be in the states **INSTALLED**, **RESOLVED**, **STARTING**, **ACTIVE**, **STOPPING** or **UNISTALLED**.



- The lifecycle of a bundle can be managed by the API of the OSGi Framework.

Management of the bundle lifecycle I

BundleActivator

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

One can specify actions a bundle should perfom when it is started and stopped. To this end the following interface BundleActivator is to be implemented:

```
public interface BundleActivator{
   public void start(BundleContext context)
      throws Exception;
   public void stop(BundleContext context)
      throws Exception;
}
```

Management of the bundle lifecycle II

BundleActivator

The implementing class (only one per bundle allowed) must have a public, no-argument constructor.

Activator class of bundle "SomeBundle":

```
package org.example;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class HelloWorldActivator implements
   BundleActivator {
     public HelloWorldActivator() {}
     public void start(BundleContext context)
        throws Exception {
          System.out.println("Hello OSGi-World!");
     }
     public void stop(BundleContext context)
        throws Exception {
          System.out.println("Goodbye OSGi-World!");
     }
}
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Management of the bundle lifecycle III

BundleActivator

MANIFEST.MF of bundle "SomeBundle":

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Bundle with bundle activator
Bundle-SymbolicName: org.example
Bundle-Version: 1.0.0
Import-Package: org.osgi.framework;version="1.5.0"
Bundle-Activator: org.example.HelloWorldActivator
```

Management of the bundle lifecycle IV

BundleActivator

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

A bundle can use a BundleActivator to store the given BundleContext:

```
...
public class HelloWorldActivator implements
  BundleActivator {
    private BundleContext bundleContext;
    public void start(BundleContext context)
      throws Exception {
        this.bundleContext = context;
    }
...
}
```

Management of the bundle lifecycle V

BundleContext

- The BundleContext object represents the interface between all bundles and the OSGi Framework.
- This object provides methods to
  - install a new bundle:

```
public Bundle installBundle(String location)
  throws BundleException
```

  - access all installed bundles:

```
public Bundle[] getBundles()
```

  - (de-)register listeners on bundles.
  - (de-)register services a bundle provides.
  - request services of other bundles.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Management of the bundle lifecycle VI

Bundle

- Every bundle that is installed within the OSGi framework is represented by an object of type Bundle.
- This object provides methods to manipulate the lifecycle of the corresponding bundle:

```
public void start() throws BundleException
public void stop() throws BundleException
public void update() throws BundleException
public void uninstall() throws BundleException
```

Management of the bundle lifecycle VII

Bundle

Example:

```
...
private BundleContext bundleContext;

// called by start method of activator class
public void setBundleContext(BundleContext context) {
  this.bundleContext = context;
}

public void installAndStartABundle(String location) {
  try {
    Bundle bundle = bundleContext.installBundle(location);
    bundle.start();
    } catch (BundleException e) {
      e.printStackTrace();
    }
}
 ...
```

Management of the bundle lifecycle VIII
BundleListener

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- To be able to react to a changed bundle state the interface BundleListener has to be implemented:

```
public interface BundleListener
  extends EventListener{
    public void bundleChanged(BundleEvent event);
}
```

- The BundleContext object provides a methods to (de-)register a BundleListener:

```
public void addBundleListener(BundleListener
  listener)
public void removeBundleListener(BundleListener
  listener)
```

- When the state of any bundle changes, the OSGi framework calls the method bundleChanged.

Management of the bundle lifecycle IX

BundleListener

Implementation of a BundleListener:

```
public class ReportChange implements BundleListener {
  public void bundleChanged(BundleEvent event) {
    System.out.println(event.getBundle() + "changed its state");
  }
}
```

Registration of a BundleListener:

```
...
public class HelloWorldActivator implements
  BundleActivator {
    private BundleContext bundleContext;
    public void start(BundleContext context)
      throws Exception {
        this.bundleContext = context;
        ReportChange reportChange = new ReportChange();
        context.addBundleListener(reportChange);
    }
...
}
```

Services I

- A **service** is a simple Java object contained in a bundle.
- Services are registered at a central **Service Registry** where other bundles can request it.
- The **Service Registry** is part of the OSGi Framework.

Services II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

To work with a service, the following steps are necessary:

Service Registry

1. Register the service

2. Request the service

5. Deregister the service

4. Release the service

3. Use the service

- MyServiceBundle
  - META-INF
    - MANIFEST.MF
  - org.example.service.activator
    - ServiceBundleActivator.java
  - org.example.service.implementation
    - GreetingImplementation.java
  - org.example.service.interfaces
    - GreetingInterface.java

- UsingServiceBundle
  - META-INF
    - MANIFEST.MF
  - org.example.service.helloworld
    - UsingServiceActivator.java

Register the service I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- The bundle implementing the service must create the service and register this service object via the `BundleContext` at the Service Registry:

```
public ServiceRegistration registerService(String
    name, Object service, Dictionary properties)
```

- The service object is registered under a specific name (usually the name of the interface that the service implements).
- `Dictionary` is a Java class that maps keys to values. It can be used to describe properties of the service.

Register the service II

Bundle "MyServiceBundle" registers the service

Activator class of bundle "MyServiceBundle":

```
package org.example.service.activator;
...
public class ServiceBundleActivator implements BundleActivator {
  private ServiceRegistration registration;
  public void start(BundleContext context) throws Exception {
    GreetingImplementation gi = new GreetingImplementation();
    registration = context.registerService
      (GreetingInterface.class.getName(), gi, null);
  }
  public void stop(BundleContext context) throws Exception {...}
}
```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Request, use and release the service I

- The `BundleContext` provides methods to request and release a service:
  - Another bundle can request the registered service by its specific name:

    ```
    public ServiceReference getServiceReference
        (String name)
    ```

  - By means of the returned `ServiceReference`, a reference to the service object can be requested:

    ```
    public Object getService
        (ServiceReference reference)
    ```

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Request, use and release the service II

- To enable the OSGi Framework to manage which bundles are using which services, a service has to be released when it is not used any more:

  ```
  public boolean ungetService
    (ServiceReference reference)
  ```

  The returned `boolean` value is `false` if the bundle never used the service or the service was already deregistered.

- A service object can be used by different bundles at the same time.

Request, use and release the service III

"UsingServiceBundle" requests, uses and releases the service of bundle "MyServiceBundle"

Activator class of bundle "UsingServiceBundle":

```
package org.example.service.helloworld;
...
public class UsingServiceActivator implements BundleActivator {
  public void start(BundleContext context) throws Exception {
    ServiceReference reference = context.getServiceReference
      (GreetingInterface.class.getName());
    GreetingInterface gi =
      (GreetingInterface)context.getService(reference);
    gi.sayHello();
    context.ungetService(reference);
  }
  public void stop(BundleContext context) throws Exception {...}
}
```

Deregister the service I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- When the service should not be available any more, the service can be deregistered by the bundle that registered the service.
- This is done by the `ServiceRegistration` object that the method `registerService` returned:
  `public void unregister()`

Deregister the service II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Bundle "MyServiceBundle" deregisters the service

Activator class of bundle "MyServiceBundle":

```
package org.example.service.activator;
...
public class ServiceBundleActivator implements BundleActivator {
  ServiceRegistration registration;
  public void start(BundleContext context)
    throws Exception {...}
  public void stop(BundleContext context) throws Exception {
    registration.unregister();
  }
}
```

Dynamic services I

- Services are **dynamic**, i.e. they can be registered or deregistered at any time.
- The interface `ServiceTrackerCustomizer` acts as a service listener:

```
public Object addingService
  (ServiceReference reference)
public void modifiedService
  (ServiceReference reference, Object service)
public void removedService
  (ServiceReference reference, Object service)
```

Dynamic services II

Implementation of a `ServiceTrackerCustomizer`:

```
public class ReportServiceChange
    implements ServiceTrackerCustomizer {
  private BundleContext context;
  public ReportServiceChange(BundleContext context) {
    this.bundleContext = context;
  }
  public Object addingService(ServiceReference reference) {
    System.out.println(reference.getBundle.getSymbolicName()
      + "was registered"); } }
    return context.getService(reference);
  }
  public void modifiedService(ServiceReference reference,
    Object service) {...}
  public void removedService(ServiceReference reference,
    Object service) {...}
}
```

Dynamic services III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- To register a service listener, a bundle must create a `ServiceTracker`:

```
public ServiceTracker(BundleContext context,
  String name,
  ServiceTrackerCustomizer customizer)
```

- The constructor takes the name of the service that should be monitored for changes.
- The `ServiceTracker` object calls the corresponding methods of the `ServiceTrackerCustomizer` when the service is registered, deregistered or one of its properties changes.

Dynamic services IV

Registration of a `ServiceTrackerCustomizer`:

```
...
private ServiceTracker tracker;
public void start(BundleContext context) throws Exception {
  ReportServiceChange reportServiceChange =
    new ReportServiceChange(context);
  tracker = new ServiceTracker(context,
    GreetingInterface.class.getName(), reportServiceChange);
  tracker.open(); // to start the ServiceTracker
  }
...
```

OSGi Standard Services

OSGi Standard Services

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- **OSGi Standard Services** (OSGi Alliance (2010a)):
  - are based on the OSGi Framework
  - offer an API for different recurring problems
- Over 20 OSGi Standard Services are defined:
  - **Declarative Services**
  - Event Admin Service
  - Http Service
  - ...

Declarative Services I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- In large applications, the service model of the OSGi Framework has some drawbacks:
  - **Start-up time**:
    Instantiation and registration of many services takes too much time.
  - **Memory usage**:
    For every registered service, all associated classes and objects are loaded in memory.
  - **Complexity**:
    Because services can be registered and deregistered at any time, the programming model is complex.

Declarative Services II

- **Declarative Services** address these problems by introducing **service components** which
  - are not activated until the service provided by the service component is requested for the first time.
  - are not activated until all services required by the service component are available.

Service Component

A **service component** is defined in a bundle and consists of

- a **component class**:
  - simple Java class
  - must have a public, no-argument constructor
  - can implement the methods `activate(ComponentContext)` and `deactivate(ComponentContext)` to specify actions that should be performed when the component is (de-)activated
- a **component description**
  - description of the component as an XML document
  - additional line in `MANIFEST.MF`: `Service-Component: OSGI-INF/component-description.xml`

The **Service Component Runtime** creates service components and manages their lifecycle.

Example: A simple service component

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component class:

```
package org.example.simplecomponent;
import org.osgi.service.component.ComponentContext;
public class SimpleComponent {
   protected void activate(ComponentContext context) {
      System.out.println("activate");
   }
   protected void deactivate(ComponentContext context) {
      System.out.println("deactivate");
   }
}
```

Component description:

```
<?xml version="1.0"?>
<component name="simpleComponent">
  <implementation class=
    "org.example.simplecomponent.SimpleComponent"/>
</component>
```

Delayed Component I

- An instance of a service component can be registered as an OSGi service.
- This is done by adding the XML element `service` to the component description:

```
<service>
<provide interface="...">
</service>
```

- `<provide interface="...">` is used to specify the name the service should be registered under.
- A service component that provides a service is not activated until the service is requested for the first time.
- Such a service component is called a **delayed component**.

Delayed Component II

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

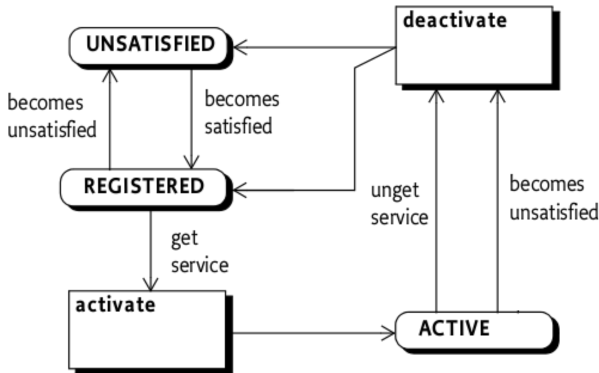
Component Interaction

Component Specification

Provisioning and Assembly

References

Lifecycle of a delayed component:



A service component is satisfied as soon as its dependencies can be resolved.

Example: A service component as a service I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Service Interface:

```
package org.example.simplecomponent;
public class SimpleService {
   public void sayHello();
}
```

Component class:

```
package org.example.simplecomponent;
import org.osgi.service.component.ComponentContext;
public class SimpleComponent implements SimpleService {
   public void sayHello() {
      System.out.println("Hello!");
   }
}
```

Example: A service component as a service II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component description:

```
<?xml version="1.0"?>
<component name="simpleComponent">
  <implementation class=
    "org.example.simplecomponent.SimpleComponent"/>
  <service>
    <provide interface=
      "org.example.simplecomponent.SimpleService"/>
  </service>
</component>
```

Immediate Component I

- A service component can use services registered by other bundles or service components.
- This is done by adding the XML element `reference` to the component description:

```
<reference
  name="..."
  interface="..."
  bind="..."
  unbind="..."
/>
```

- `name`: The local name of the reference.
- `interface`: The name the service is registered under.
- `bind`: The name of the method that is used to assign the service to the component.
- `unbind`: The name of the method that is used to remove the service from the component.

Immediate Component II

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

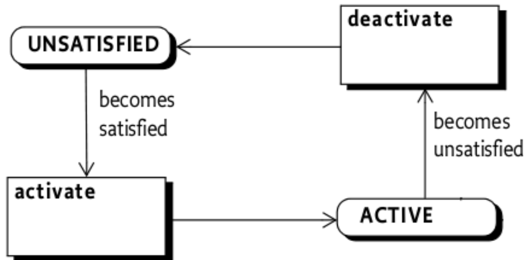
Component Interaction

Component Specification

Provisioning and Assembly

References

- A service component that uses services is activated as soon as all requested services are available.
- Such a service component is called an **immediate component**.
- Lifecycle of an immediate component:



Example: A service component uses a service I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component class:

```
package org.example.hellocomponent;
import org.osgi.service.component.ComponentContext;
public class HelloComponent {
  private SimpleService service;
  protected void setService(SimpleService service) {
    this.service = service;
  }
  protected void unsetService(SimpleService service) {
    this.service = null;
  }
  protected void activate(ComponentContext componentContext) {
    sayHello();
  }
}
```

Example: A service component uses a service II

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

Component description:

```
<?xml version="1.0"?>
<component name="helloComponent">
  <implementation class=
    "org.example.hellocomponent.HelloComponent"/>
  <reference
    name="SimpleService"
    interface="org.example.simplecomponent.SimpleComponent"
    bind="setService"
    unbind="unsetService"
  />
</component>
```

SWK

JJ+HS

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Advantages of service components

- **Delayed activation of services**:
  Services that are provided by service components will be registered at the Service Registry when the implementing bundle is started. But no service instance is created until the service is requested for the first time. This reduces **start-up time** and **memory usage**.
- **Resolution of service dependencies**:
  The Service Component Runtime resolves all service dependencies. It instantiates and activates a service component that uses services not until all necessary services are available. Therefore, no service listeners have to be implemented. This reduces **complexity**.

What have we learned? I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- OSGi defines a dynamic component model for Java.
- In OSGi, components are called bundles. Bundles
  - consist of Java packages and an additional file with descriptive information (e.g. about exports and imports).
  - have a lifecycle that can be controlled by the OSGi Framework API.
  - can implement services which are registered at the Service Registry where other bundles can request them
- OSGi Standard Services offer an API for different recurring problems, like Declarative Services which reduce start-up time, memory usage and complexity when working with services.

What have we learned? II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

	Description	Management	Listener
Bundle	OSGi component	`BundleActivator` (`start` and `stop` methods), `BundleContext`, `Bundle`	`BundleListener`
Service	Java object contained in a bundle	Service Registry	`ServiceTracker-Customizer`, `ServiceTracker`
Service Component	Java object and component description contained in a bundle	Service Component Runtime, `activate` and `deactivate` methods	
Delayed Component	Service component which provides a service	Service Component Runtime, `activate` and `deactivate` methods	
Immediate Component	Service component which uses services	Service Component Runtime, `activate` and `deactivate` methods	

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

A Process for Specifying Component-Based Software

by Cheesman and Daniels (2001)

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Motivation

- Major challenge in software engineering today: **manage change**
- For Cheesman and Daniels, the objective of component reuse is of less importance.
- Aim: provide advice, guidance, and examples for modeling enterprise-scale component systems.

Architectural layers

Application

System

User Interface
Creates what the user sees.
Handles UI logic.

User Dialog
Dialog logic.
Transient state corresponds to the dialog.
Can sometimes be used with multiple UIs.

client part

server part

System Services
Operations are new transactions.
Can be used with a variety of user dialogs or batch.
Components correspond to business systems.
No dialog or client-related state.

Business Services
Components correspond to stable business types or groups.
Operations can be combined with others in a transaction.
Usually have associated databases.

Example components in the layers

User Interface

Java server page displaying "reserve car" dialog

User Dialog

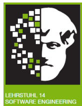
Java bean holding "reserve car" dialog state

System Services

EJB session bean supporting "auto rental" system interface

Business Services

EJB entity bean representing a rental contract



Realization vs. usage contracts

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

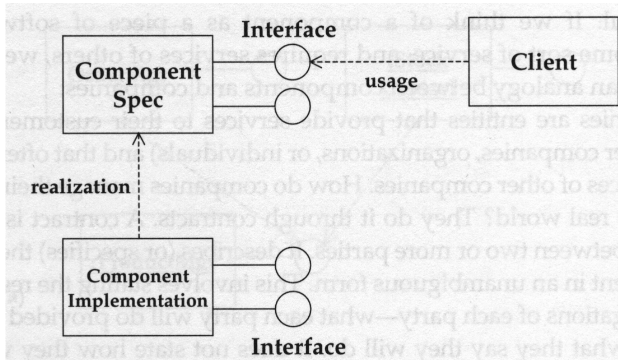
Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References



Interface- vs. component specification

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Interface	Component Specification
A list of operations	A list of supported interfaces
Defines an underlying logical information model	Defines the relationships between the information models of different interfaces
Represents the contract with the client	Represents the contract with the implementer
Specifies how operations affect or rely on the information model	Defines the implementation and runtime unit
Describes local effects only	Specifies how operations must be implemented in terms of usage of other interfaces

Workflow of the overall development process

Business requirements

Requirements

Existing assets

Use case models

Business concept models

Technical constraints

Components

Use Case models

Specification

Provisioning

Assembly

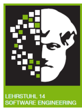
Component specs & architectures

Applications

Test

Tested applications

Deployment



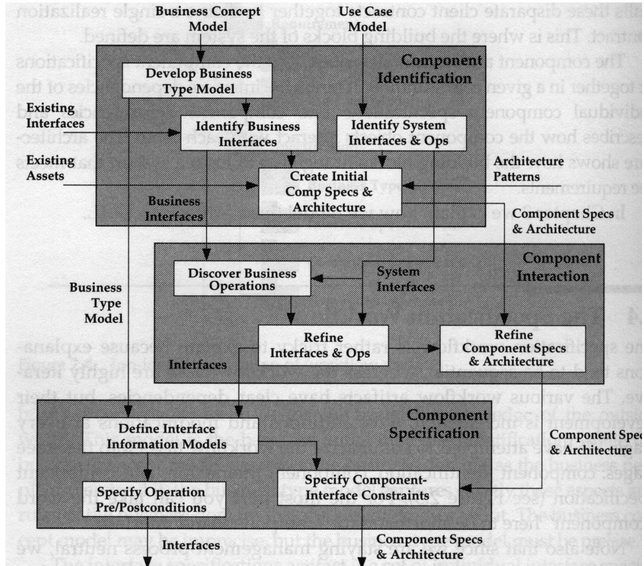
Stages of the process

Requirements
Requirements Definition → Chapter 4

Specification
Component Identification → Chapter 5
Component Interaction → Chapter 6
Component Specification → Chapter 7

Provisioning → Chapter 8
Assembly → Chapter 8

Stages of the specification workflow



Models to be produced

- Requirements
  - Business Concept Model
  - Use Case Model
- Specification
  - Business Type Model
  - Interface Specifications
  - Component Specifications
  - Component Architecture

Notations used

Use Case Diagram

Use Case Diagrams

Requirements

Business Concept Model Diagram

Business Concept Model

Use Case Model

Class Diagram

Specification

Business Type Model Diagram

Class Diagram

Interface Specification Diagrams

Business Type Model

Interface Specifications

Interface Responsibility Diagram

Component Specification Diagrams

Component Specifications

Component Architecture

Component Architecture Diagram

Interactions

Component Interaction Diagrams

Collaboration Diagram

Package Diagram

Summary of UML extensions

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Component Specification Concept	UML Construct	Stereotype
Component Specification	Class	«comp spec»
Interface Type	Type (Class «type»)	«interface type»
Comp Spec offers Interface Type	Dependency	«offers»
Business Concept	Class	«concept» (optional)
Business Type	Type (Class «type»)	«type»
Structured Data Type	Type (Class «type»)	«datatype»
Interface Information Type	Type (Class «type»)	«info type» (often omitted)
Parameterized Attribute	Operation	«att»
Operation requiring a new transaction	Operation	«transaction»

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Requirements Definition

Requirements definition: overview

1. Business process
2. Business concept model
3. System envisioning
4. Use cases
   - 4.1. Actors and roles
   - 4.2. Use case identification
   - 4.3. Use case descriptions
   - 4.4. Quality of service

Business process

- Business process to be supported must be understood
- Its description is **not** a statement of the requirements for the IT system (software)
- Notation: e.g., UML activity diagrams

Example of a business process

SWK

JJ+HS

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

Running example: hotel reservation

enquiry/
Check availability
[else]
[suitable room]
Make reservation
Confirm reservation
Wait for event
customer arrives/
Take up reservation
cancel request/
Cancel reservation
amendment request/
Amend reservation
no-show/
Process no-show
Notify billing system

Business concept model

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Expresses domain knowledge about the application domain; thus, it is not related to software.
- Does not need to be tightly scoped to the problem
- Notation: UML class diagrams

Example of a business concept model

Hotel Chain
1
1..*
Hotel
1
1..*
Clerk
*
contactedHotel
1
1
*
Customer
1
*
Reservation
*
allocation
0..1
1..*
Room
1
contactAddress
0..1
Address
1
*
*
1
RoomType
1
0..1
1
Bill
Payment
0..1

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

System envisioning I

Define the software boundary; make clear which functions are the responsibility of the software.

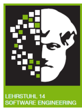
Example:

> *A hotel reservation system is required that will allow reservations to be made for any hotel in the chain. At present each hotel has its own, incompatible, system. Reservations can be made by telephone to a dedicated central reservation center, by telephone direct to a hotel, or via the Internet. A major advantage of the new system will be the ability to offer rooms at alternative hotels when the desired hotel is full.*

System envisioning II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

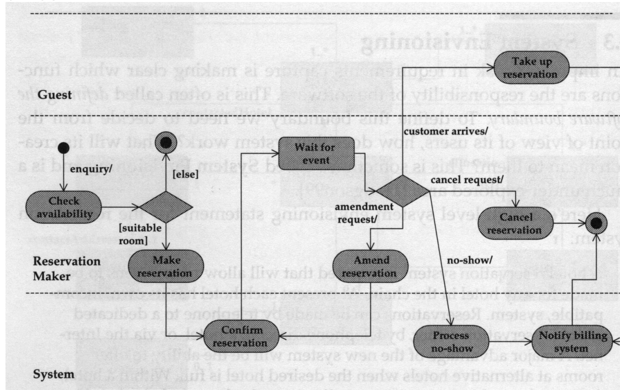
Within a hotel, facilities for making reservations will exist at the front desk, in the office, and at the concierge's desk. Each hotel has a reservation administrator who is responsible for controlling reservations at the hotel, but any authorized user may make a reservation. The target time for making a reservation by telephone or in person is three minutes. To speed up the process, details of previous customers will be stored and made available.



Use cases

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Allocate responsibility for the business process steps. Notation: swim lanes.

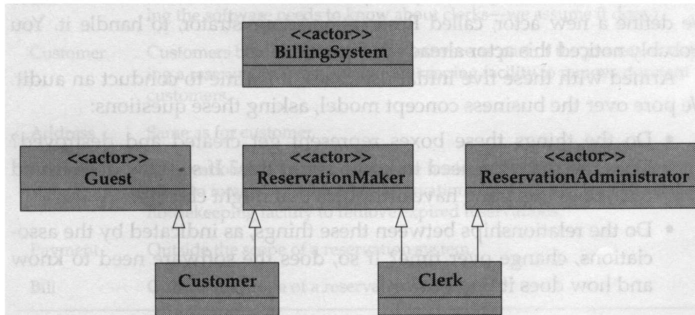


Note: responsibility decisions have a profound effect on the shape of the resulting software. They are often taken too quickly.

Actors and roles

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Actors are **roles** that initiate and control the steps assigned to them, even though the software may play a part in these steps.
- To be flexible, generalization relations can be introduced.



Use case identification I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Use cases

- describe the interaction of actors with the software
- are a functional specification of the software
- define the boundary between the software and its environment
- **describe the interaction that follows from a single business event**

Use case identification II

Hotel example: five events (corresponding to five use cases)

1. Make Reservation (covering Check Availability, Make Reservation, and Confirm Reservation steps)
2. Cancel Reservation
3. Amend Reservation (covering Amend Reservation and Confirm Reservation)
4. Take Up Reservation (covering Take Up Reservation and Notify Billing System)
5. Process No-Show (covering Process No-Show and Notify Billing System)

Use case identification III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Discussion

- Not turning up is a bit of a noevent. A business rule must define when the no-show event is generated, e.g. no arrival until 8 p.m.
- The processing of no-shows can either be triggered by a clock and be performed automatically, or be initiated by a user (which is chosen here).
- Therefore, the use case is renamed Process No-Shows, because it deals with all reservations that meet the no-show business rule.
- But who is the corresponding actor?
  Introduce ReservationAdministrator (see above)

Use case identification IV

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Audit

Considering the business concept model, answer the following questions:

- about the classes
  - Do the things these boxes represent get created and destroyed?
  - Does the software need to know about this?
  - If so, how does it find out?
  - Does this thing have attributes that might change?
- about the associations
  - Do the relationships between these things change over time?
  - If so, does the software need to know and how does it find out?

Use case identification V

HotelChain	The requirement is a reservation system for a single chain, so we never create or destroy chains.
Hotel	Hotels might be added or removed, albeit infrequently, so we will need use cases for these events.
Room	Rooms might be added or removed, so we need use cases for these events.
RoomType	Room types might be added or removed, so we need use cases for these events.
Clerk	Clerks will come and go, so we need use cases for these events. (Assuming the software needs to know about clerks—we assume it does.)
Customer	Customers become known to the software as part of the process of making a reservation. We need a housekeeping facility to remove dormant customers.
Address	Same as for customer.
Reservation	Reservations are created during the business process. We want the software to remember completed reservations for a year, so we will need a housekeeping facility to remove expired reservations.
Payment	Outside the scope of a reservation system.
Bill	Outside the scope of a reservation system.

Use case identification VI

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

HotelChain-Hotel	Never changes.
Hotel-Room	Never changes (can't move a room from one hotel to another).
Hotel-Clerk	Clerks can move from one hotel to another but we decide that the software won't support this. The details will need to be re-entered.
Hotel-Customer	Not to be maintained in the software.
Hotel-Reservation	Can be changed as part of reservation amendment.
Customer-Address	Never changes (but the details of an address might change).
Customer-Reservation	Never changes.
Reservation-RoomType	Can be changed as part of reservation amendment.
Reservation-Bill	Out of the scope of the system.
Reservation-Room	An interesting one! We decide (after much consultation with the domain experts) that this association is made when the customer arrives to take up his or her reservation. There will be no preallocation of rooms.
Bill-Payment	Out of the scope of the system.
RoomType-Room	Never changes (can't change a single room to a double).

Use case identification VII

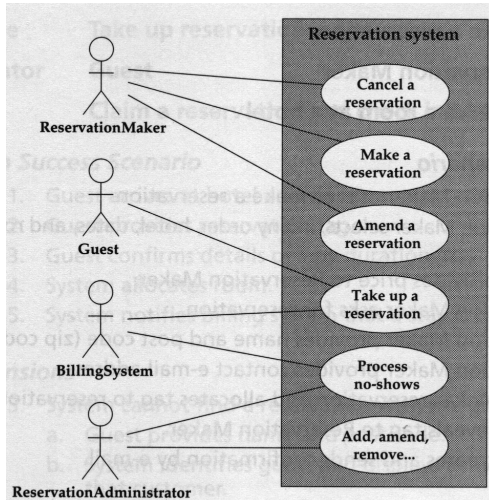
We assume that all the things in our model might have attributes that can change, so the full list of uses cases, so far as we know now, is as follows:

- Make Reservation
- Cancel Reservation
- Amend Reservation
- Take Up Reservation
- Process No-Shows
- Add/Amend/Remove Hotel
- Add/Amend/Remove Room
- Add/Amend/Remove Room Type
- Add/Amend/Remove Clerk
- Amend Customer
- Remove Dormant Customers
- Amend Address
- Remove Old Reservations

Use case identification VIII

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Resulting use case diagram



Use case descriptions I

For each use case, describe main success scenario, then add extensions and variations.

Name **Make a reservation**

Initiator **Reservation Maker**

Goal **Reserve a room at a hotel**

Main Success Scenario

1. Reservation Maker asks to make a reservation.
2. Reservation Maker selects, in any order, hotel, dates, and room type.
3. System provides price to Reservation Maker.
4. Reservation Maker asks for reservation.
5. Reservation Maker provides name and post code (zip code).
6. Reservation Maker provides contact e-mail address.
7. System makes reservation and allocates tag to reservation.
8. System reveals tag to Reservation Maker.
9. System creates and sends confirmation by e-mail.

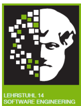
Extensions

3. Room not available.
   a. System offers alternatives.
   b. Reservation Maker selects from alternatives.

3b. Reservation Maker rejects alternatives.
   a. Fail

4. Reservation Maker declines offer.
   a. Fail

6. Customer already on file (based on name and post code).
   a. Resume 7.



Use case descriptions II

Name	**Take up reservation**
Initiator	**Guest**
Goal	**Claim a reservation and check in to the hotel**

Main Success Scenario

1. Guest arrives at hotel and claims a reservation.
2. Guest provides reservation tag.
3. Guest confirms details of stay duration, room type.
4. System allocates room.
5. System notifies billing system that a stay is starting.

Extensions

3. System cannot find a reservation with the given tag.
   a. Guest provides name and post code.
   b. System identifies guest and displays active reservations for that customer.
   c. Guest selects the reservation.
   d. Resume 4.

3. The reservation tag refers to a reservation at a different hotel.
   a2. Fail

3c. No active reservations at this hotel for this customer.
   a. Fail

Variations

At 4 the Guest may wish to change stay details.

Use case descriptions III

If we were to continue with the other uses cases, we would find that the extensions in Take Up Reservation occur in several use cases. As a convenience, we can factor this out into a separate use case:

Name **Identify reservation**

Initiator **Included only**

Goal **Identify an existing reservation**

Main Success Scenario

1. Actor provides reservation tag.
2. System locates reservation.

Extensions

2. System cannot find a reservation with the given tag.
   a. Actor provides name and post code.
   b. System displays active reservations for that customer.
   c. Actor selects the reservation.
   d. Stop.
2. The reservation tag refers to a reservation at a different hotel.
   a2. Fail

2b. No active reservations at this hotel for this customer.
   a. Fail

Use case descriptions IV

We can then simplify the Take Up Reservation use case:

Name **Take up reservation**

Initiator **Guest**

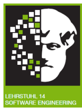
Goal **Claim a reservation and check in to the hotel**

Main Success Scenario

1. Guest arrives at hotel and claims a reservation.
2. Include Identify Reservation.
3. Guest confirms details of stay duration, room type.
4. System allocates room.
5. System notifies billing system that a stay is starting.

Extensions

3. Reservation not identified.
   a. Fail



Use case descriptions V

Final use case diagram:

Reservation system
ReservationMaker
Guest
BillingSystem
ReservationAdministrator
Cancel a reservation
Make a reservation
Update a reservation
Take up a reservation
Process no-shows
Add, amend, remove...
Identify reservation
<<include>>
<<include>>
<<include>>

Quality of service I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- We ought to add a quality of service section to each use case, stating our expectations, especially in the areas of security and performance.
- Where these requirements are system-wide, we can state them separately.
- For example, we might say:

> *Only authorized users (identified by a password) may access the reservation service, other than via the Internet.*

Quality of service II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- For the Make a Reservation use case, our quality of service statement might be

  The system must support 200 simultaneous users.

  System response to any input must not exceed 2 seconds (95 percent) for direct connections and 5 seconds (90 percent) for Internet connections.

  *The system must support (total number of rooms) * 10 active reservations, and assume 100 percent hotel occupancy.*

Summary of requirements definition

- The requirements workflow must deliver to the specification workflow a business concept model and a set of use cases.
- The business concept model lists the important concepts in the problem domain and shows the relationships between them.
- The use cases clarify the software boundary, identify the actors who interact with the software, and describe those interactions.

Component identification stage of the specification workflow I

Figure 5.1 The component identification stage of the specification workflow

Component identification stage of the specification workflow II

- Goal: create an initial set of interfaces and component specifications, hooked together into a first-cut component architecture.
- Emphasis: discovery
  - What information needs to be managed?
  - What interfaces are needed to manage it?
  - What components are needed to provide that functionality?
  - How will they fit together?
- Identify the system interfaces and system components in the system services layer.
- Identify the business interfaces and business type components in the business services layer.
- Take into account existing interfaces, databases, or components that need to be interfaced with and that may need adapting.
- Try to apply architectural patterns.

Focus of interface identification

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

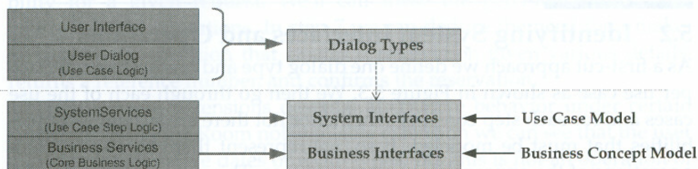


Figure 5.2 Interface inputs and correspondence to application architecture layers

- Process is concerned with the UI-independent aspects of an application, corresponding to the server side of things.
- Refine business concept model (representing human's eye view) into business type model (representing software's eye view).
- Use business type model to develop business interfaces.
- The implementations of components supporting these interfaces form the core business logic.

Runtime behavior

- When a user initiates a use case, the use case logic causes the appropriate UI to be created and displayed.
- The user is guided through the use case steps by the use case logic.
- Whenever the use case logic needs information to display or needs to notify the system of a user action, it calls the appropriate operation in the use case step logic.
- This operation, in turn, uses operations defined in the core business logic to perform its function.

Note: A component may only invoke operations on its own level or in a level below itself.

Identifying system interfaces and operations

- Identify one dialog type and one system interface per use case.
- Then go through each use case and for each step consider whether or not there are any system responsibilities that must be modeled.
- If so, represent them as one or more operations on the appropriate system interface.

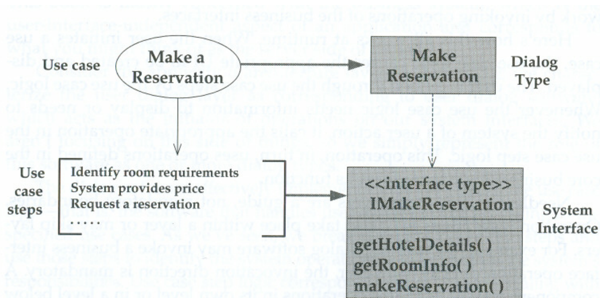


Figure 5.3 Use cases map to system interfaces

Example: Make a reservation

Introduction
Patterns
Components
Design by contract
Components a OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Name	**Make a reservation**
Initiator	**Reservation Maker**
Goal	**Reserve a room at a hotel**

Main Success Scenario

1. Reservation Maker asks to make a reservation.
2. Reservation Maker selects, in any order, hotel, dates, and room type.
3. System provides price to Reservation Maker.
4. Reservation Maker asks for reservation.
5. Reservation Maker provides name and post code (zip code).
6. Reservation Maker provides contact e-mail address.
7. System makes reservation and allocates tag to reservation.
8. System reveals tag to Reservation Maker.
9. System creates and sends confirmation by e-mail.

Extensions

3. Room not available.
   a. System offers alternatives.
   b. Reservation Maker selects from alternatives.

3b. Reservation Maker rejects alternatives.
   a. Fail

4. Reservation Maker declines offer.
   a. Fail

6. Customer already on file (based on name and post code).
   a. Resume 7.

- Define initial system interface called IMakeReservation.
- Step 2: system must allow to get details of different hotels (getHotelDetails()).
- Step 3: Price and availability for a given request must be provided (getRoomInfo()).
- Step 7: operation makeReservation() needed that creates a reservation, returns a reference number, and confirms the reservation.

Results of identifying system interfaces and operations

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Parameters of the operations are defined later when considering the component interactions.
- The interfaces we have defined at system level are specific to that system and will not typically be reusable by different systems.
- Reuse of interfaces across systems is the purpose of the business interfaces, to be discussed next.

Identifying business interfaces

The business interfaces are abstractions of the information that must be managed by the system. The process for identifying them is as follows:

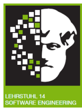
1. Produce a scoped copy of the business concept model as the business type model.
2. Refine the business type model and specify any additional business rules with constraints.
3. Identify **Core Business Types**.
4. Create business interfaces for core types and then add them to the business type model.
5. Refine the business type model to indicate business interface responsibilities.
6. Check that the defined interfaces align with any overriding policies, such as those defined in a corporate component architecture.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Create the business type model

- The business type model is represented by a UML class diagram, like the concept model, but its purpose is different.
- Whereas the concept model is simply a map of the information of interest in the problem domain, the business type model contains the specific business information that must be held by the system being specified.
- The business type model is initially created by copying the concept model and adding or removing elements until its scope is correct.

Note: The business type model must be a precise model, because it is the base from which the business interfaces will emerge.



Example I

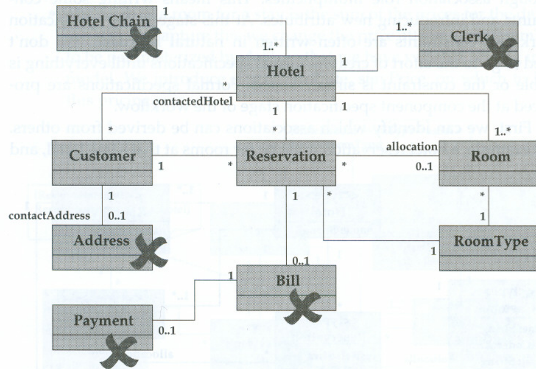


Figure 5.6 Scoping the business type model

Example II

- Eliminate the HotelChain type because the system shall support only a single chain of hotels.
- Eliminate the Hotel-Customer association (see use case definition phase).
- Eliminate Payment and Bill because they are the domain of a separate billing system.
- Eliminate Clerk and Address to keep the example simpler.

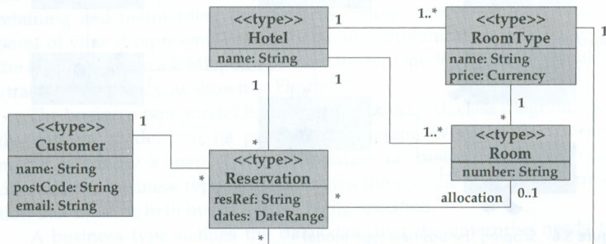


Figure 5.7 Initial business type

Define business rules I

- Add any additional required business rules to the simple ones captured directly through association role multiplicities.
- This means writing some constraints and introducing new attributes.

Example:

- Identify which associations can be derived from others:
  - A hotel reservation must be for rooms at that same hotel, and the type of room specified must be available at that same hotel.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Define business rules II

- Availability rules:
  - A room is available if the number of rooms reserved at all dates in the requested range is less than the number of rooms.
    Introduce new parameterized attribute available(DateRange) for RoomType, on which to hang this rule.
  - You can never have more reservations for a date than rooms (no overbooking).
- Pricing rules
  - The price of a room for a stay is the sum of the prices for the days in the stay.
    Change price attribute on RoomType to be parameterized by date.
    Introduce new attribute stayPrice, on which to hang this rule.

Define business rules III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Adding the extra attributes allows us to write these rules in OCL.

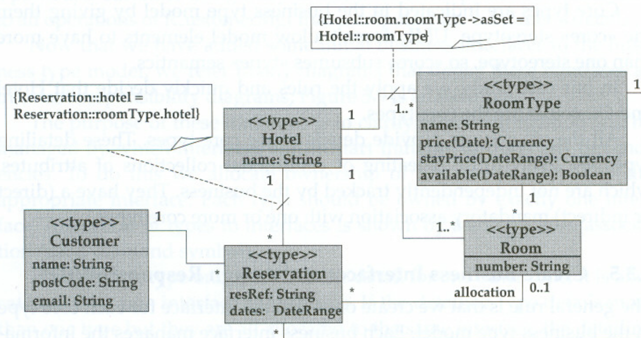


Figure 5.8 Business type model

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Identify core types

- The purpose of identifying core types is to start thinking about which information is dependent on which other information, and which information can stand alone.
- A core business type is a business type that has independent existence within the business.
- Example: core types are Hotel and Customer.

Create business interfaces and assign responsibilities I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- General rule: create one business interface for each core type of the business type model.
- Each business interface manages the information represented by the core type and its detailing types.
- Naming convention: IxxxMgt

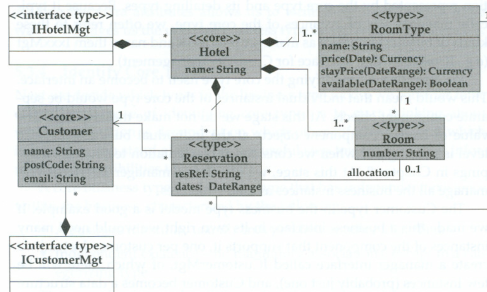


Figure 5.9 Interface responsibility diagram of the business type model

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Create business interfaces and assign responsibilities II

- Each type should be owned by exactly one interface (composition relation).
- Where to allocate Reservation (provides details to both Hotel and Customer)?
- Decision: allocate Reservation to Hotel; mark association between Reservation and Customer to be navigable only toward customer.

Allocating responsibility for associations I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- When an association exists between types managed by different interfaces, this is an *inter-interface association*.
- The association between Reservation and Customer is such an association.
- A decision has to be made where this information will be recorded.
- Inter-interface associations are a specific form of dependency, which contradict the high-level goal to reduce dependencies.
- Therefore: try to avoid two-way references between interfaces.

Allocating responsibility for associations II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Decision: Reservation references Customer, and Customer is independent of Reservation.
- Association is navigable in only one direction.

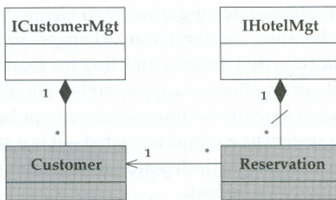


Figure 5.10 Assigning reference direction

1. How this is achieved in the implementation is, of course, a totally separate issue.

Creating initial interface specifications I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- The system interfaces that we created earlier, which are not part of the business type model, form an initial set of interface specifications that subsequent stages will refine directly.
- The business type model and the business interfaces are internal workflow artefacts.
- Once we are happy with the interface responsibility diagram, we create another set of business interfaces in the interface specifications package, corresponding to the business interfaces we created in the business type model.
- We will further work on those interfaces in the component specification phase.

Creating initial interface specifications II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

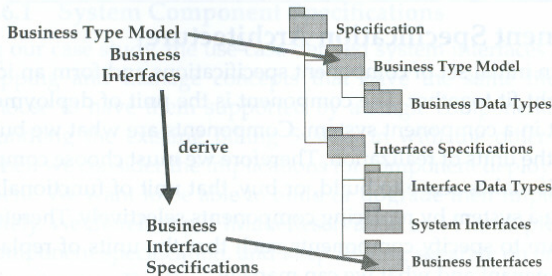


Figure 5.11 Package structure detail

Existing interfaces and systems

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Add to the interface specifications package any additional interfaces that are part of the environment into which the software will be deployed.
- In particular, are there any existing interfaces that we are obliged to use?
- Are there any systems with which we need to interface, but which are outside the specific scope of the given development project?
- Example: billing system. Its interfaces are added to the set of system interfaces.

Component Specification Architecture

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- We now create an initial set of component specifications and form an idea of how they might fit together.
- We must choose components in such a way that it makes sense to build or to buy that unit of functionality.
- In most cases, we will create a separate component specification for each interface specification that we have identified.
- Multiple interfaces on one component can be considered if
  - The concepts represented by the different interfaces have the same lifetime.
  - The interactions between the interfaces are complex, frequent, or involve large amounts of data.
  - We want to keep component granularity at a reasonable size.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

System component specifications I

- In our case study, the use-case-driven system interfaces are strongly overlapping and manage concepts that have the same lifetimes.
- We therefore put IMakeReservation and ITakeUpReservation on one component.
- However, IBilling is kept separate.
- The reservation system makes use of IBilling, so we add the dependency between them.
- We also add interface dependencies on ICustomerMgt and IHotelMgt, although we don't know if these really exist at this stage.
- We will validate these when we study the component interactions.

System component specifications II

SWK

JJ+HS

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

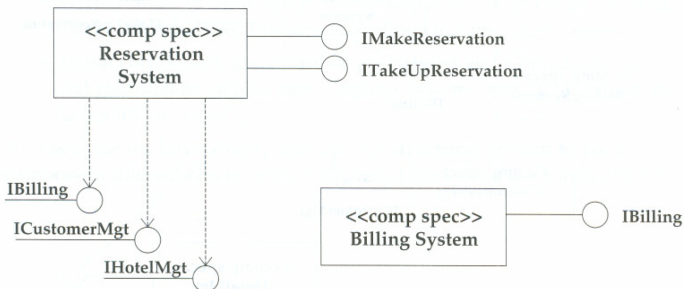


Figure 5.12 System component specifications

Business component specifications

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- For the business interfaces, our starting point is one component per interface.
- Since the manager interfaces were created to manage instances of core business types and their associations, they are concerned with information that is managed independently.
- Result:

An initial architecture I

- Now we have an initial set of component specifications, including their supported interfaces and their interface dependencies.
- Since we don't have any interfaces being offered by more than one component specification in our example, we can bind the interface dependencies of the component specifications directly onto their corresponding component specification interfaces.

An initial architecture II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Result:

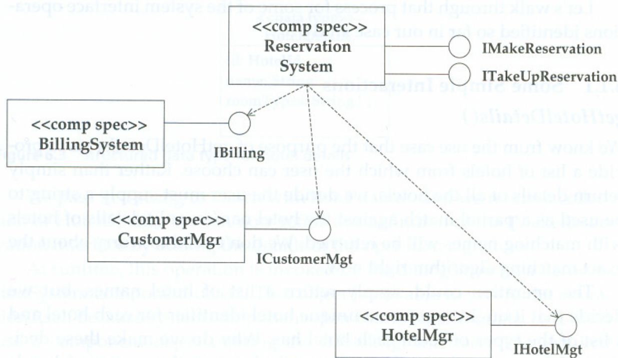


Figure 6.2 Initial component architecture

Summary of component identification I

Main principles:

- The system interfaces correspond to use cases, and their operations are derived from use case steps.
- A business type model is developed representing the system's eye view of the business concept model. Business rules are captured on the business type model as constraints. The business type model is an internal workflow artifact, which is useful to maintain.
- Business interfaces are discovered by identifying core types in the business type model and creating interfaces to manage them and their details.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Summary of component identification II

- Initial business interface specifications are created by copying the business type model interfaces. These interfaces are refined in subsequent stages.
- Initial component specifications are defined and formed into an initial component architecture. Existing systems and architectures are taken into account.

Component interaction

- The component identification gives us an initial set of interfaces and components with which to work.
- Now we will decide how the components will work together to deliver the required functionality.

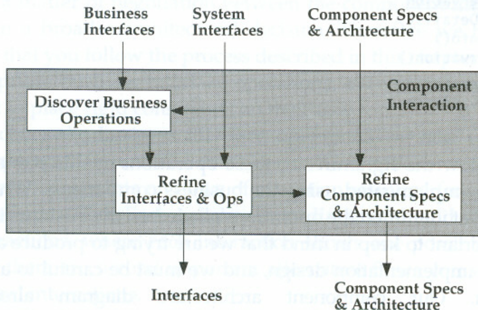


Figure 6.1 The component interaction stage of the specification workflow

Discovering business interfaces I

- We have identified the operations of the system interfaces.
- Example: Interface IMakeReservation has the operations getHotelDetails(), getRoomInfo(), and makeReservation().
- We do not know the signatures of these operations at this point, nor how they will be implemented using business components.
- We haven't even identified the operations needed on the business interfaces.
- Our component architecture diagram tells implementers of ReservationSystem that they must use the ICustomerMgt and IHotelMgt interfaces.

Discovering business interfaces II

Procedure for discovering business operations:

- Take each system interface operation and draw one or more collaboration diagrams that trace any constraints on flows of execution resulting from an invocation of that operation.
- Each collaboration diagram should show one or more interactions, where each interaction shows one possible execution flow.
- So if there are several important flows, one will need to draw several interactions.

getHotelDetails() I

- Input: string to be used as a partial match against the hotel names.
- Output: collection of hotel details

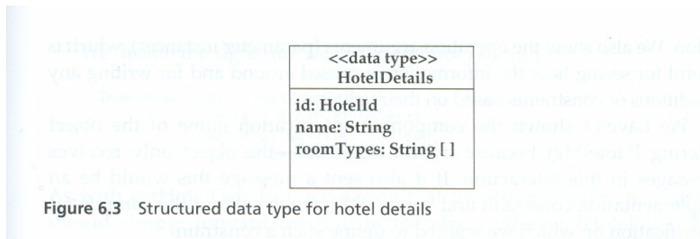


Figure 6.3 Structured data type for hotel details

```
IMakeReservation::getHotelDetails(
    in match:  String):  HotelDetails []
```

getHotelDetails() II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Collaboration diagram:

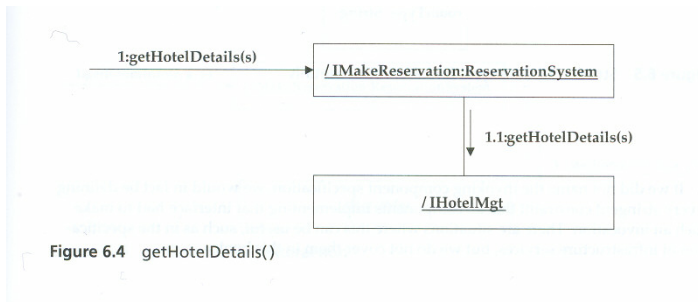


Figure 6.4 getHotelDetails()

(Notation: objectname/rolename:classifiername)

getRoomInfo()

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

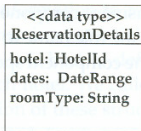


Figure 6.5 Structured data type for reservation details

```
IMakeReservation::getRoomInfo(
in res: ReservationDetails,
out availability: Boolean, out price: Currency)
```

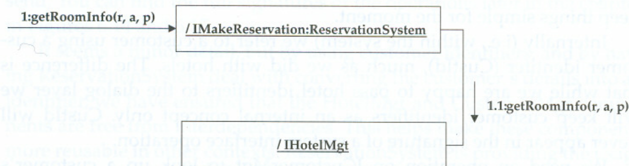


Figure 6.6 getRoomInfo() interaction

makeReservation(): breaking dependencies I

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

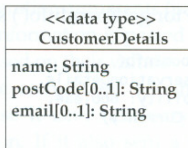


Figure 6.7 Structured data type for customer details

```
IMakeReservation::makeReservation(
   in res:  ReservationDetails,
   in cus:  CustomerDetails, out resRef:  String):
                                            Integer
```

where the return value indicates the outcome of the operation

- 0: Success.
- 1: Customer does not exist, no new record could be created, because post code and/or e-mail address were not provided.

makeReservation(): breaking dependencies II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- 2: No post code was provided, and the name matches more than one customer.

We need an operation on ICustomerMgt to look up a customer's details and return his or her CustId, so we invent one:

```
ICustomerMgt::getCustomerMatching(
   in cusD: CustomerDetails,
   out cusID: CustId):  Integer
```

where 0: success; 1: customer does not exist; 2: as above.

makeReservation(): breaking dependencies III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Which of our components is going to call that operation?

- The HotelMgr component is responsible for storing the association between reservations and customers.
- The HotelMgr and CustomerMgr components are independent of each other!
- Therefore, we cannot let the ReservationSystem component forward the makeReservation() call to the HotelMgr and let it get on with it, because then HotelMgr would have to use CustomerMgr.
- Instead, the ReservationSystem is going to have to do this.

makeReservation(): breaking dependencies IV

/ ICustomerMgt

1:makeReservation(r, c, rr)

1.1:getCustomerMatching(c, id) {result=0}

1.3:notifyCustomer(id, s)

{where s includes rr etc}

/ IMakeReservation:ReservationSystem

1.2:makeReservation(r, id, rr)

/ IHotelMgt

Figure 6.8 makeReservation() interaction (existing customer)

Maintaining referential integrity

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- We haven't said how many component objects there will be at runtime.
- Example: ReservationSystem will always use the same business component objects.
- Expressed using a component specification diagram.

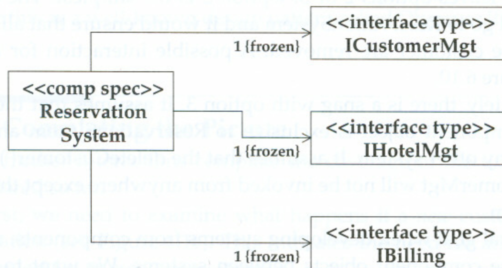


Figure 6.9 Constraints on the component object architecture

Controlling intercomponent references I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Options for allocating responsibility that intercomponent references are valid (example: deletion of a customer):

1. Allocate responsibility to the component object storing the reference.
   Example: make sure that all requests to delete customers are sent to the HotelMgr component.
2. Allocate responsibility to the component object that owns the target of the reference.
   Example: this would be CustomerMgr.
3. Allocate responsibility to a third object, usually higher up in the call chain.
   Example: ReservationSystem.
4. Permit, and tolerate, references to become invalid.
5. Disallow the deletion of information.

Controlling intercomponent references II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

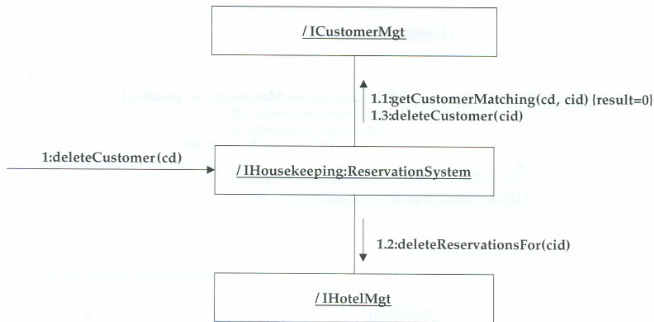


Figure 6.10 Interaction for referential integrity option 3

Disadvantage of option 3: assumes that the CustomerMgr component is object is exclusive to the ReservationSystem.

If this assumption cannot be made, option 2 must be used. Realization using Observer design pattern.

Completing the picture

Introduction

Patterns

Components

Design by contract

Components and OO

Java Beans

OSGi

Component Spec. Proc.

Requirements Definition

Component Identification

Component Interaction

Component Specification

Provisioning and Assembly

References

- What happens if a *new* customer makes a reservation?
- Need for an operation on ICustomerMgt to create a new customer.

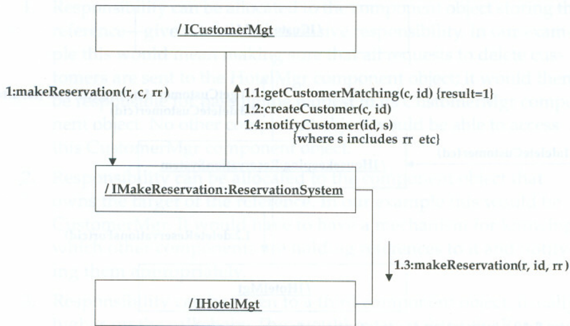
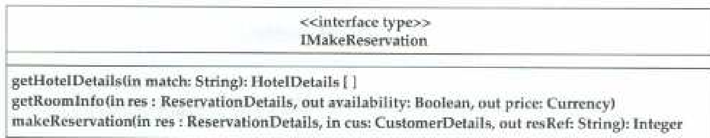
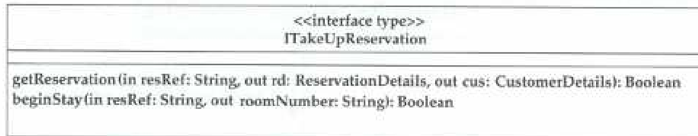


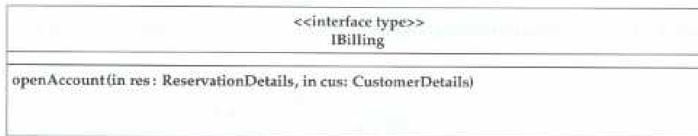
Figure 6.11 makeReservation() interaction (new customer)

Considering the Take Up Reservation Use case also gives rise to new operations on IHotelMgt and ICustomerMgt.

System interfaces with operation signatures

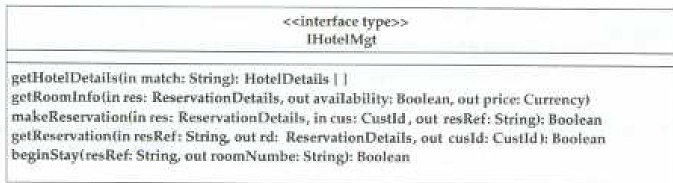






Business interfaces with operation signatures

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

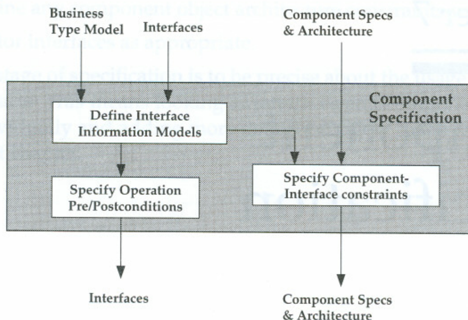


Summary of component interaction

- Develop interaction models for each system interface operation.
- Discover business interface operations and their signatures.
- Refine responsibilities.
- Define any component architecture constraints you need.

Component specification

- A usage contract is defined by an interface specification.
- A realization contract is defined by a component specification.
- Component specifications are primarily groupings of interfaces.
- Component (and interface) specification is the final stage of the specification workflow.



Specifying interfaces

- An interface is a set of operations.
- An operation represents a fine-grained contract between a client and a component object.
- To express the contract, we need a construct that describes the state of a component object.

Operation specification

An operation specifies the individual action that a component object will perform for a client. This has a number of facets:

- The input parameters: specifying the information provided or passed to the component object.
- The output parameters: specifying the information updated or returned by the component object.
- Any resulting change of state of the component object.
- Any constraints that apply (precondition).

However, operation specifications on interfaces do not include information about interactions between the component object performing the operation and other component objects that are required, in a specific implementation, to complete the operation.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Interface information models I

- We need to represent the state of the component on which the interface depends.
- To do this, each interface has an interface information model.
- All changes to the state of the component object caused by a given operation can be described in terms of this information model definition.

Interface information models II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Example:

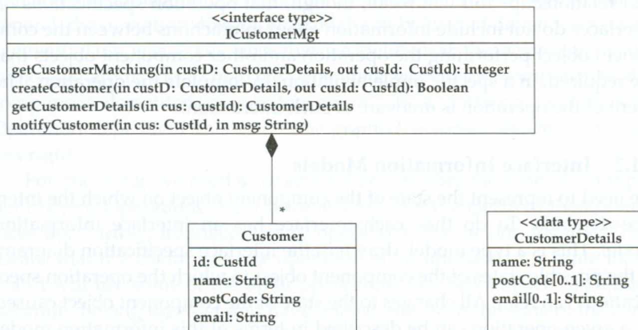


Figure 7.2 Interface specification diagram for the ICustomerMgt interface

Pre- and postconditions I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Each operation has a pre- and a postcondition.
- These can be defined precisely using OCL.
- The OCL expressions can refer to the operation parameters, the operation result, and the state of the component object (as defined by the interface information model).
- The OCL expressions cannot refer to anything else.

Pre- and postconditions II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Example:

```
context ICustomerMgt::getCustomerDetails
          (cus: CustId): CustomerDetails

pre: -- cus is a valid customer
customer->exists(c: Customer | c.id = cus)

post:
-- the details returned match the details
-- of the customer whose id is cus
-- find the customer

let theCust: Customer = customer->
select(c: Customer| c.id = cus)->asSequence()->first() in
    result.name = theCust.name and
    result.postCode = theCust.postCode and
    result.email = theCust.email
```

From business type model to interface information model I

Result of component identification phase:

<<interface type>>
IHotelMgt

<<core>>
Hotel
name: String

<<type>>
RoomType
name: String
price(Date): Currency
stayPrice(DateRange): Currency
available(DateRange): Boolean

<<core>>
Customer
name: String
postCode: String
email: String

<<type>>
Reservation
resRef: String
dates: DateRange

<<type>>
Room
number: String

allocation

<<interface type>>
ICustomerMgt

1 * 1..* 1..* 0..1

Figure 7.3 Case study interface responsibility diagram

From business type model to interface information model II

- Start by making a copy of the business type model in the interface's package.
- Delete types, associations, and attributes that are not needed.
- When a type owned by one interface refers to a type owned by another, the referenced type (Customer, in this case) appears in the interface information models of both interfaces.
- However, it need not look the same in both interfaces.
- For example, the Customer type in IHotelMgt only needs the customer id.

From business type model to interface information model III

Result:

<<interface type>>
IHotelMgt

getHotelDetails(in match: String): HotelDetails [ ]
getRoomInfo(in res : ReservationDetails, out availability: Boolean, out price: Currency)
makeReservation(in res : ReservationDetails, in cus :CustId , out resRef: String): Boolean
getReservation(in resRef: String, out rd: ReservationDetails, out cusId: CustId): Boolean
beginStay(resRef: String, out roomNumber: String): Boolean

Reservation
resRef: String
dates: DateRange
claimed: Boolean

Hotel
id: HotelId
name: String

Room
number: String

Customer
id: CustId

RoomType
name: String
available(during: DateRange): Boolean
price(on: Date): Currency
stayPrice(for: DateRange): Currency

* 1 1..* 0..1 allocation

Figure 7.4 Interface specification diagram for IHotelMgt

Invariants

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- An invariant is a constraint attached to a type that must be held true for all instances of the type.
- Many invariants can be expressed graphically, using UML notation (e.g., multiplicities).
- In some cases it isn't possible or convenient to use the graphical notation. Use OCL instead.

Example:

```
context Reservation
-- a reservation is claimed
-- if it has a room allocated to it
inv:  claimed = allocation->notEmpty()
```

Snapshots

A useful technique when writing pre- and postconditions is to draw "before" and "after" instance diagrams and to highlight the state changes that occur.

before

:Hotel
{id=4}

:RoomType
{name=single}

:RoomType
{name=double}

:Reservation
{refRef=H51}

:Customer
{id=38}

after

:Hotel
{id=4}

:Reservation
{resRef=A77}

:RoomType
{name=single}

:RoomType
{name=double}

:Reservation
{refRef=H51}

:Customer
{id=92}

:Customer
{id=38}

Figure 7.5 "Before" and "after" snapshot instance diagrams for IHotelMgt::makeReservation()

Specification of IHotelMgt::makeReservation I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
context IHotelMgt::makeReservation
   (res: ReservationDetails, cus: CustId, resRef: String)
   : Boolean

pre:
-- the hotel id and room type are valid
hotel->exists(h | h.id = res.hotel
      and h.room.roomType.name->includes(res.roomType))

post:
result implies
-- a reservation was created
-- identify the hotel
```

Specification of IHotelMgt::makeReservation II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
let h: Hotel = hotel->select(x | x.id = res.hotel)
->asSequence()->first() in

-- only one more reservation now than before
h.reservation->size() - h.reservation@pre->size() = 1

-- identify the reservation
and let r: Reservation = h.reservation->
    select(y: Reservation| not h.reservation@pre->
        includes(y))->asSequence()->first() in

-- return number is number of the new reservation
r.resRef = resRef and
```

Specification of IHotelMgt::makeReservation III

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

```
-- other attributes match
r.dates = res.dates and
r.roomType.name = res.roomType
and not r.claimed and
r.customer.id = cus
```

Specifying system interfaces I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Until now, we have discussed a systematic way of moving from the business type model to the information models of the business interfaces.
- For the system interfaces, we take a similar approach.
- As with any other interface, the interface information model of a system interface needs to contain just enough information for the operations to be specified.
- This will be a subset of the business type model.
- Note that the existence of an interface information model does not imply that an implementation of the interface must store the information persistently. In fact, system interfaces rarely have persistent storage.

Introduction

Patterns

Components
- Design by contract
- Components and OO
- Java Beans
- OSGi
- Component Spec. Proc.
  - Requirements Definition
  - Component Identification
  - Component Interaction
  - Component Specification
  - Provisioning and Assembly

References

Specifying system interfaces II

Figure 7.6 Interface specification diagram for IMakeReservation

Note that the information model for IMakeReservation does not require the room number attribute, so it has been removed.

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

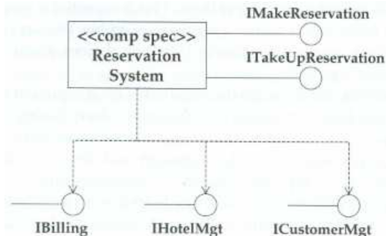
Specifying components

- The interface specifications discussed so far deal with the usage contract – the contract between a component object and its clients.
- Now we consider the additional specification information that the component implementer and assembler need to be aware of, especially the dependencies of a component on other interfaces.
- This information forms the component specification.

Offered and used interfaces

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- For every component specification we need to say which interfaces its realizations must support, see architecture diagram of component identification phase.
- Now, we must dissect that diagram into pieces specific to each component specification.
- We also need to confirm any constraints concerning which other interfaces are to be used by a realization (dependency arrows in architecture diagram).



Scoping interactions I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Constraints how a particular operation must be implemented are defined in interactions.
- Component interactions define specification-level constraints. All component realizations must respect them.
- This is essential if we aim to be able to replace components within a complex component assembly.
- The interactions that make up the constraints on component specifications are typically fragments of the interactions we drew during operation discovery.
- They begin with a component object receiving a message, and only show the direct interactions from that component.

Scoping interactions II

anotherComponent: B

1:doIt (a, b)

theComponentBeingSpecified/IW: A

1.1:doIt (a, b)

aComponentSupporting/IX: C

1.1.1: doIt (a, b)

aComponentSupporting/IY

Figure 7.11 Scoping an interaction

Inter-interface constraints

We may want to express constraints concerning the relationships between interface information models. This concerns

- how offered interfaces relate to each other
- how offered interfaces relate to used interfaces.

Offered interfaces

- The Reservation System component offers the IMakeReservation and ITakeUpReservation interfaces.
- Both these interfaces have a Reservation information type.
- Since the two interfaces are specified completely independently, we cannot assume that that both reservation types are the same.
- This has to be expressed explicitly.

```
context ReservationSystem
-- constraints between offered interfaces
IMakeReservation::hotel = ITakeUpReservation::hotel
IMakeReservation::reservation =
                     ITakeUpReservation::reservation
IMakeReservation::customer = ITakeUpReservation::customer
```

where a formal definition of "=" depends on the two information types involved.

Offered and used interfaces

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Note that the existence of an interface information model does not imply that implementations of the interface will store the information.
- Instead, they obtain the information from the business components.
- Therefore, we write constraints that require the elements of the interface information models to match up.

```
context ReservationSystem
-- constraints between offered and used interfaces
IMakeReservation::hotel = IHotelMgt::hotel
IMakeReservation::reservation =
                        IHotelMgt::reservation
IMakeReservation::customer = ICustomerMgt::customer
```

Factoring interfaces I

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- Each interface has its own interface information model, which is often only slightly different from the model of another interface.
- Sometimes it is possible to simplify things by refactoring the interfaces, especially by introducing new abstract interfaces that act as super-types of other interfaces, holding common interface information model elements, and, sometimes, definitions of common operations.
- In some cases it may even be practical to simply merge system interfaces together and do not bother with subtyping. This may be appropriate when the corresponding use cases have the same actors.

Factoring interfaces II

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

Example: factor out the common elements of the information models from IMakeReservation and ITakeUpReservation and place them in a new interface called IReservationSystem:

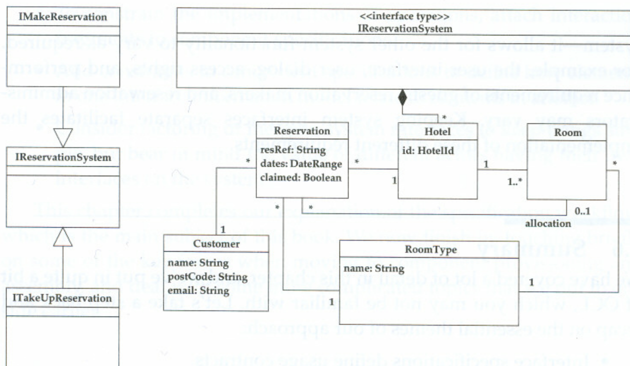


Figure 7.12 Refactoring interfaces

Factoring interfaces III

SWK

JJ+HS

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

The interface information model for IMakeReservation then merely extends the inherited types, adding extra attributes that are required.

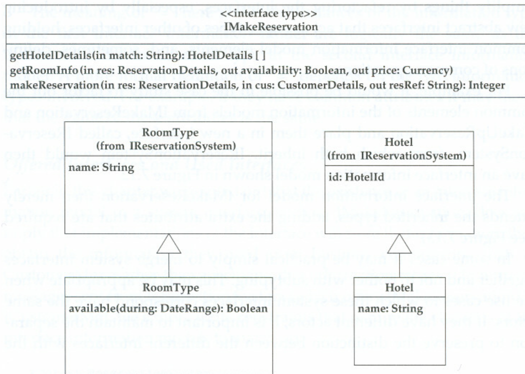


Figure 7.13 IMakeReservation after factoring out IReservationSystem

Summary of component specification I

- Interface specifications define usage contracts.
- Component specifications define realization contracts.
- An interface is specified by a set of operation specifications that operate on an interface information model.
- The interface information model must contain just enough information to allow the operations to be specified. It cannot refer to anything outside the interface.
- First-cut interface information models can be derived systematically from the business type model.
- Each operation is specified using a pre- and postcondition pair.

Summary of component specification II

- OCL can be used to express invariants and operation pre- and postconditions.
- Component specifications include specifications of the interfaces offered and used.
- To constrain the implementations of operations, attach interaction fragments to component specifications.
- Add constraints to component specifications to define how elements in one interface information model relate to elements in another.
- Consider factoring or merging system interfaces to keep things simple, but bear in mind the value of different actors having their own interfaces on the system.

Provisioning and Assembly

Introduction
Patterns
Components
Design by contract
Components and OO
Java Beans
OSGi
Component Spec. Proc.
Requirements Definition
Component Identification
Component Interaction
Component Specification
Provisioning and Assembly
References

- In the specification workflow, we have been working in a technology-independent way.
- Provisioning means to provide component implementations, either by directly implementing the specifications or by finding an existing component that fits the specification.
- Assembly pulls the components together, using the component architecture for the software to define the overall structure and the individual pieces, and adding user interface and dialog logic.

Issues in provisioning

- A component realizes a component specification and an interface realizes an interface type.
- The realizations are performed in some target technology.
- We must consider what mappings need to take place for these two key realizations, between the technology-neutral and the technology-specific level.
- Main issues:
  - Operation parameter type, kind (in/out/inout/return), and reference restrictions
  - Exception and error handling mechanisms (implementing the contracts)
  - Interface inheritance and support restrictions
  - Operation sequence
  - Interface properties
  - Object creation mechanisms
  - Raising events

Bibliography I

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley, Boston, MA, USA, 1st edition.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

Cheesman, J. and Daniels, J. (2001). *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley.

Coplien, J. O. (1992). *Advanced C++ Programming Styles and Idioms*. Addison-Wesley.

Bibliography II

Coplien, J. O. (1998). C++ idioms. http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html (last visit: May 27th, 2009).

D'Souza, D. and Wills, A. C. (1998). *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading.

Heineman, G. T. and Councill, W. T. (2001). *Component-Based Software Engineering*. Addison-Wesley.

Bibliography III

Heisel, M., Santen, T., and Souquières, J. (2002). Toward a formal model of software components. In *Proc. 4th International Conference on Formal Engineering Methods*, pages 57–68. Springer.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall International, 2nd edition.

OSGi Alliance (2010a). *OSGi Service Platform Release 4 Version 4.2 Compendium Specification*. http://www.osgi.org/Download/Release4V42.

OSGi Alliance (2010b). *OSGi Service Platform Release 4 Version 4.2 Core Specification*. http://www.osgi.org/Download/Release4V42.

Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component Software*. Pearson Education. Second edition.

Bibliography IV

Wütherich, G., Hartmann, N., Kolb, B., and Lübken, M. (2008). *Die OSGi Service Platform: Eine Einführung mit Eclipse*. dpunkt.