Advanced Topics in OOA&D

Jürgen Börstler
jubo@cs.umu.se
http://www.cs.umu.se/~jubo

http://www.cs.umu.se/kurser/TDBC31/

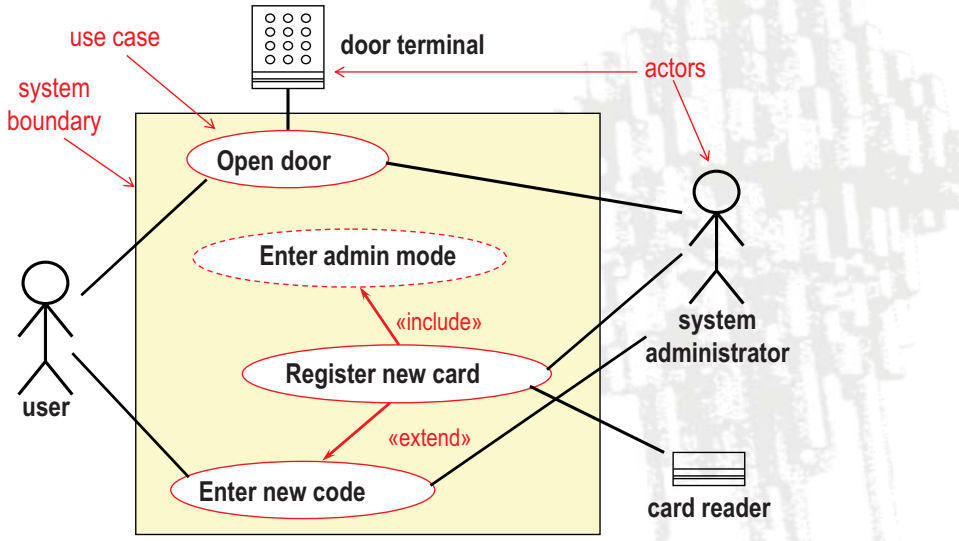
Contents

More Advanced Linguistic Analysis á la KISS ([Krist 94])

{Subject} {Predicate} {Direct object} [{Preposition} {Indirect object}]

- The subject carries out (controls) an action
- The direct object undergoes this action
- The action results in a state change in the direct object
- The indirect object collaborates to perform the action
- The predicate contains or describes the action
- The preposition indicates the type or kind of collaboration (relationship)

A Use Case Model

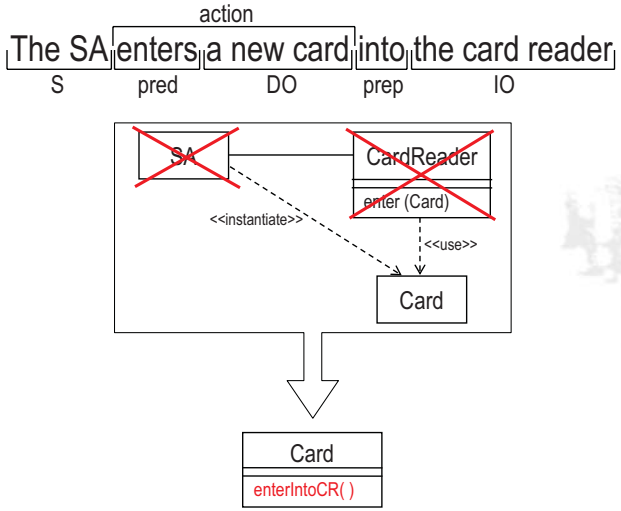


An Example Use Case

use case: Register New Card

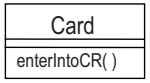
actor:	System administrator (SA)
summary:	...
preconditions:	The system is idle
actions:	
	1. Enter admin mode
	2. The SA enters a new card into the cardreader
	3. The system validates the card
	4. The system registers the card in the database
	5. Enter new code
postconditions:	The new card is registered in database
	The new card has a valid code
exceptions:	3: Card invalid

Example Analysis 1

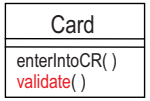


- The subject controls an action
- The direct object undergoes this action
- The action results in a state change in the direct object
- The predicate contains or describes the action
- The indirect object collaborates to perform the action
- The preposition indicates the type or kind of collaboration
- Actors are (usually) outside the scope of the system

Example Analysis 2



The system validates the card



The system registers the card in the database

- The subject controls an action
- The direct object undergoes this action
- The action results in a state change in the direct object
- The predicate contains or describes the action
- The indirect object collaborates to perform the action
- The preposition indicates the type or kind of collaboration
- Actors are (usually) outside the scope of the system

Contents

When and How (not) to Use Inheritance

- Reuse (ad hoc and planned)
- Combine behaviour
- Prototyping
- Versioning
- Parameterisation of collection types
- Commit to common interfaces (*abstract classes*)
- Build frameworks

➔ Express commonalties
➔ Extensibility

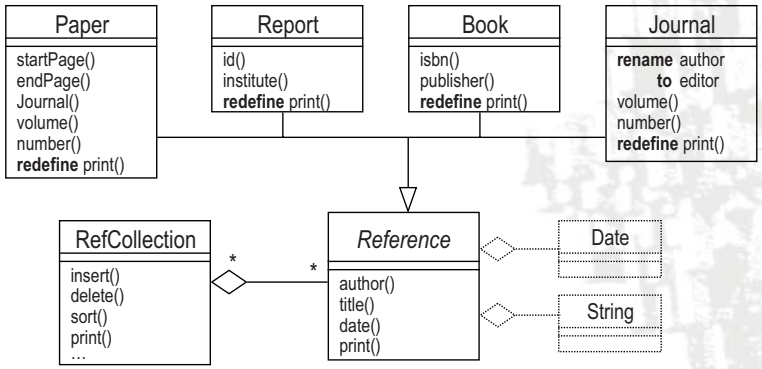
Bibliographic References—An Inheritance Example

Develop a design for the core components in a system to manage and print bibliographic references

- Different kinds of references
  - Books
  - Papers
  - Conference proceedings
  - Journals
  - Reports
  - …
- Collections of references
- Printing operations

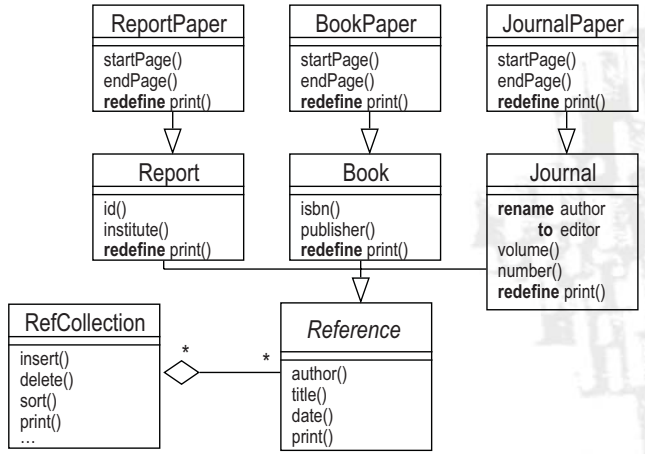
First Approach

- Check common properties



☺ Easy to add reference types
☹ Paper bound to `Journal`; Journal info twice

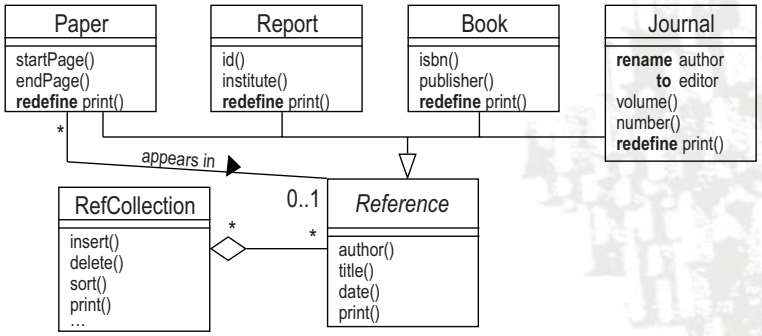
Second Approach—Uncouple `Paper` from `Journal`



☺
☹ New type requires addition of two classes

Third Approach

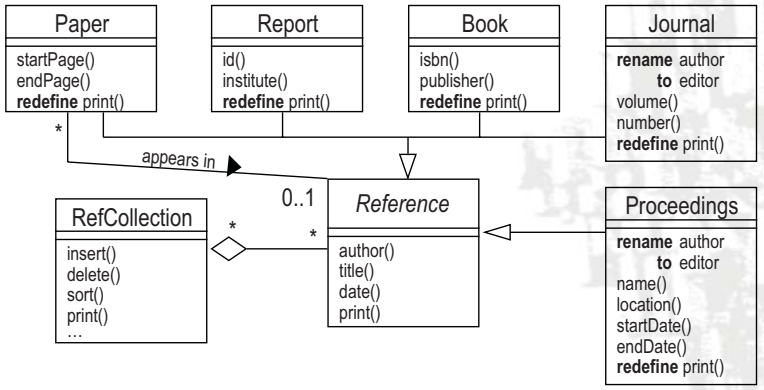
- Papers are references **and** can also be part of references



☺ Solves our problems?

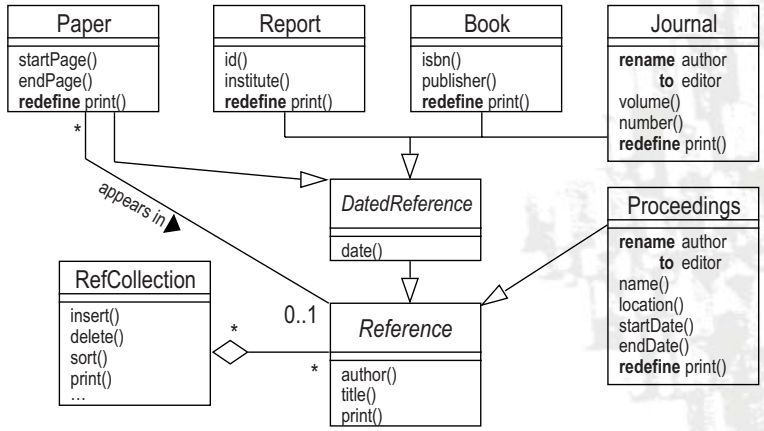
☹ (Try to add a conference proceedings)

Third Approach—Add a Reference Type



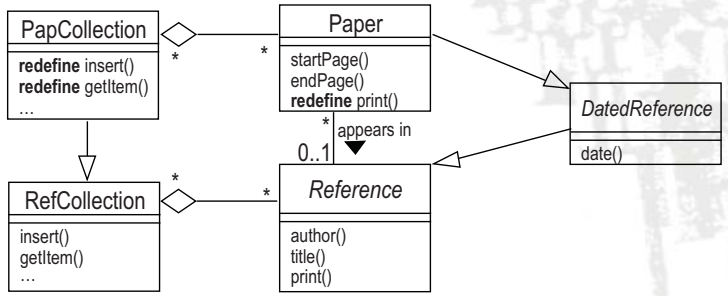
☹ There are too many dates

Refined Third Approach—Split `Reference`



Inheritance for Parameterisation

- `RefCollection` can contain objects of type `Reference` and all its subtypes
- Assume we need homogenous collections
- ➔ Can we inherit from `RefCollection`?
- ➔ Apply horizontal modification (covariance)?



Inheritance for Parameterisation is NOT Type-Save

```
// Assume we have a collection of references (aRefCollection),
// a collection of papers (aPapCollection), a reference object (aReference),
// and a paper object (aPaper)
aRefCollection.insert( aReference);      // Insert a reference into a collection of references
aRefCollection := aPapCollection;        // OK, since PapCollection is-a RefCollection
aRefCollection.insert( aReference);      // ERROR! Figure out parameter types

// OK, so what if we do not redefine any operations?
aRefCollection.insert( aReference);      // Insert a reference into a collection of references
aRefCollection.insert( aPaper);          // OK, since Paper is-a Reference
aPaper := aRefCollection.getItem();      // ERROR! Polymorphy does only work that way
```

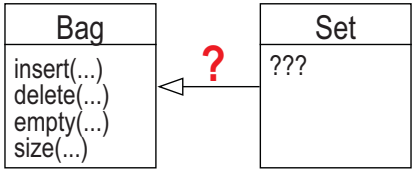
➔ Inheritance cannot replace genericity (except in theory)

➔ Use templates to parameterise collection types

Strict Inheritance

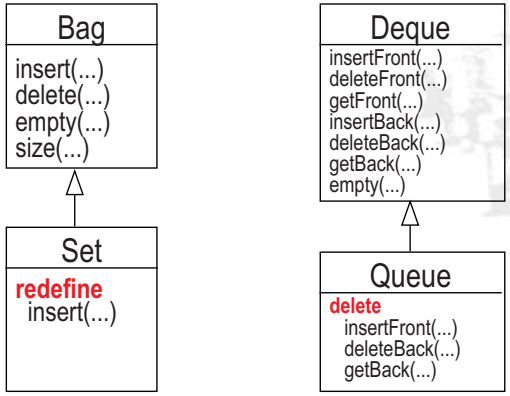
- Only extensions to inherited properties
- ➔ Easy to handle
- ➔ Inflexible
- ➔ Hinders reuse

- Example:



Non-Strict Inheritance

- Inherited properties can be changed/ rejected
- ➔ Increased flexibility and reuse

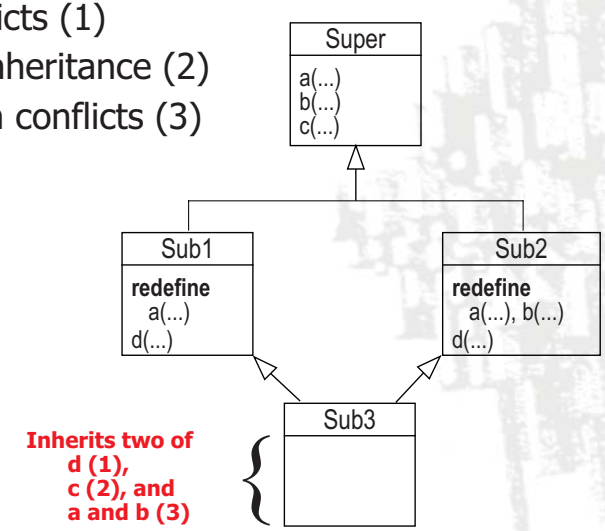


Forms of Non-Strict Inheritance

- Redefinition of inherited methods
  - Change code only
  - ➔ OK in most OO languages
  - Vertical modification (change parameter types)
    - Covariance (useful, but difficult to make type safe)
    - Contravariance (simple, but not very useful)
  - Horizontal modification (add/delete parameters)
  - ➔ Subclass is not a subtype
- Renaming
  - OK in some languages (e.g., Eiffel)
- Deletion of inherited methods
  - ➔ Subclass is not a subtype
- Attribute types must not be changed

Problems with Multiple Inheritance

- Name conflicts (1)
- Repeated inheritance (2)
- Redefinition conflicts (3)



Summary of Rules

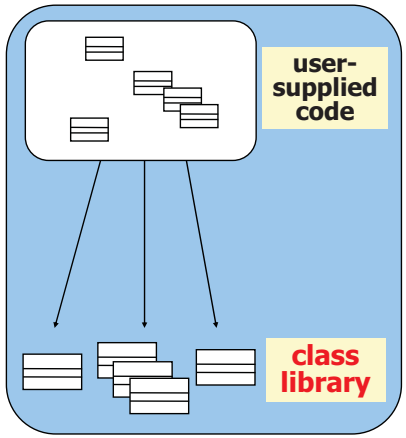
- Use abstract classes whenever possible
- Do not confuse inheritance with aggregation (is-a vs. has-a)
- Avoid adding identical behaviour in different branches of your inheritance hierarchy
- Do not add too much behaviour in one step
- Use templates to parameterise collection classes
- Use inheritance for subtyping
- Be careful with multiple inheritance

Contents

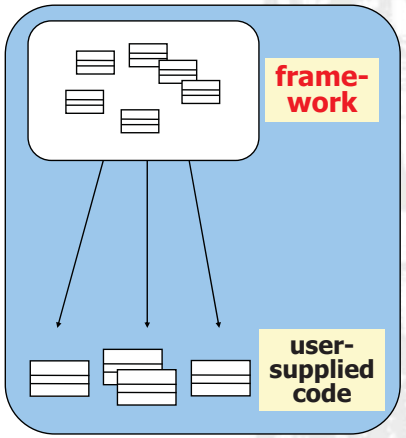
Class Libraries and Frameworks

- Class library
  - Language specific
  - Provides low-level functionality
  - Passive
  - Used by user-supplied code
- Framework
  - Language specific
  - Provides complex functionality
  - Active
  - Uses user-supplied code
- Application framework
  - Adaptable software system
  - Domain specific

Class Libraries vs. Frameworks



Users write **pro-active** code



Users write **re-active** code
(Don´t call us we´ll call you)

Contents

Design Guidelines and Patterns

What is Good Design?

- Well structured: consistent with chosen properties such as information hiding
- As simple as possible, but not simpler
- Efficient: functionality can be provided using available resources
- Adequate: meeting the stated requirements
- Flexible: "easy" to change
- Practical: provide required functionality, but not more
- Implementable using current and available technology
- Standardized: using well-defined and familiar notation(s)

Packaging Design Experiences for Reuse

- Design principles and heuristics
  - Generally accepted "rules of thumb," recommendations, and guidelines
  - Language independent
- Refactoring
  - Restructure existing code without affecting its external behaviour
- Design patterns
  - Concrete solutions to known design problems
  - Language independent
  - Common description format
- Pattern languages
  - Sets of interrelated patterns

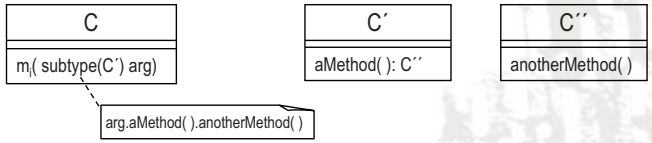
Design Principles

- Law of Demeter
- Liskov Substitution Principle
- Open-Closed Principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Single-Responsibility Principle
- Common Closure Principle

See [Martin 02] for details on most of the principles.

The Law of Demeter—Main Idea

- No class must depend on the structure of another class, like for example in

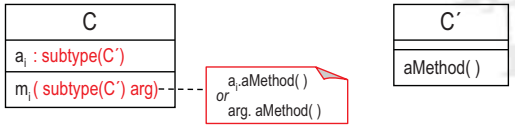


- Each method should only send messages to objects of explicitly "known" classes
- Minimise the number of acquaintance classes (only implicitly known, but called anyway)

See [LiHo 89] for details

The Law of Demeter—Acquaintance Classes

- C´ is an acquaintance class of $C.m_i$, if
  - m_i sends a message to C´, and
  - m_i has no arguments of type subtype(C´)
  - C has no instance variables of type subtype(C´)
  - (OOPS! C´ ∈ subtype(C´))



LoD—Strong Version

- A method C.m must only refer objects, which are either
  - Instantiated by this method, or
  - Instance variables of C, or
  - Arguments of C.m, or
  - Global variables, or
  - The pseudo-variable self/ this

➔ All dependencies are **explicit**

- Counter example violating the LoD (common)

```
anObject.aMethod( ).anotherMethod( )
```

returns an object of an acquaintance class

Liskov Substitution Principle

Subtypes must be substitutable for their basetypes.

- A client using an instance of a base class should still work properly when given an instance of a subclass instead
- The subclass must at least provide the same services as the superclass
- The contracts of the base class must be honoured by the subclass

Open-Closed Principle

Software entities (classes, modules, functions, …) should be open for extension, but closed for modification.

- Change behaviour by adding code and without changing existing code
- The entity can be extended to accommodate new requirements and contexts
- Existing clients are not affected by the change

Dependency Inversion Principle

Abstractions should not depend on details (implementations). Details should depend on abstractions.

- High-level entities should not depend on low-level entities
- Access instances using interfaces or abstract classes

Interface Segregation Principle

Clients should not be forced to depend on methods that they do not use.

- Many client-specific interfaces are better that one general purpose interface
- Large and general interfaces generate unnecessary dependencies, i.e. high coupling

Single Responsibility Principle

A class should only have one reason to change.

- Single, well-defined responsibility
- ➔ Loose coupling, few dependencies
- ➔ Changes stay local

Common Closure Principle

The classes in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all classes in that package and no other package.

- Closely related to Single Responsibility and Open-Closed principles, but applied to package level

Design Guidelines and Patterns

- Introduction
- Design Principles
- Refactoring
- Design Heuristics
- Design Patterns
- References

Refactoring

- Recommended XP/Agile practice
- Identify code that doesn't look right ("code smells" or "anti-patterns")
- Restructure systematically to fix problem

➔ Make code more robust wrt. changes

- There are about 30 "code smells" and about 100 refactorings
- Refactoring tools support complex restructurings

Example Code Smells

- Something has become (too) large, e.g.,
  - Long methods, many parameters
- Solution is not OO, e.g.,
  - Switch statements, no strict inheritance, "dumb" storage classes
- Things that make changes difficult, e.g.,
  - Parallel inheritance, "shotgun surgery"
- Useless or unnecessary code, e.g.,
  - Duplicate or very similar code, speculative genericity
- Unnecessary coupling
  - "Feature envy", long message chains
- Comments
  - Should be "strategic" (for clarification) only

Example Refactorings

- Rename Method: change declaration and all occurences
- Extract Method: turn a code fragment into a method
- Inline Method: reverse of Extract Method (for trivial methods that are not heavily used)
- Move Method: move a method to another class
- Hide Method: make a method private
- Introduce Parameter Object: turn a set of parameters that are used together to a new class
- …

See http://www.refactoring.com for details.

Design Guidelines and Patterns

- Introduction
- Design Principles
- Refactoring
- Design Heuristics
- Design Patterns
- References

Design Heuristics and Patterns

- Introduction
- The "God" Class Problem
- The Proliferation of Classes Problem
- Conflicting Heuristics

Riel´s OOD Heuristics

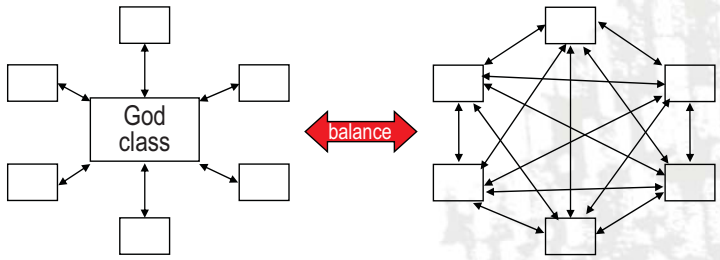
- An OO design heuristic
  - is a "rule-of-thumb"
  - is something which makes a design "feel right"
  - guides a designer
  - helps to choose from design alternatives
  - warns when it is violated
- Different design heuristics may conflict
- 61 heuristics in 8 categories

See [Riel 96] for details.

Examples for OOD Heuristics

- All data should be hidden within its class
- Most of the methods defined on a class should be using most of the data most of the time
- Do not clutter the public interface of a class with items that users of that class are not interested in or not able to use
- The interface of an application should be dependent on the model, not vice versa
- Minimise fanout in a class (#messages defined x #messages send)
- Avoid explicit case analysis on object types or attribute values
- Avoid "god" classes
- Avoid the "proliferation" of classes

God Class vs. Proliferation of Classes

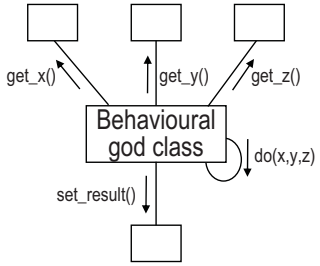


Proliferation of classes

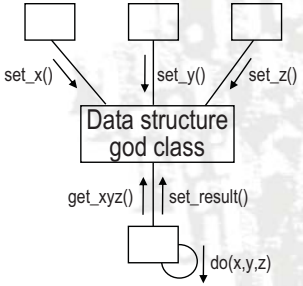
- System intelligence should be distributed, but to which extent
  - ? Complexity
  - ? Maintainability
  - ? Fault tolerance

Variations of the God Class Problem

Central brain class to control behaviour

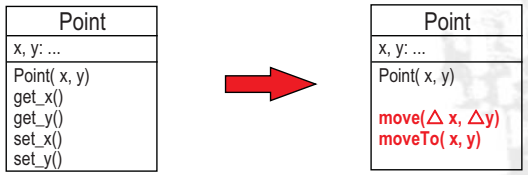


Global data class to store all system data



Coping with the God Class Problem 1

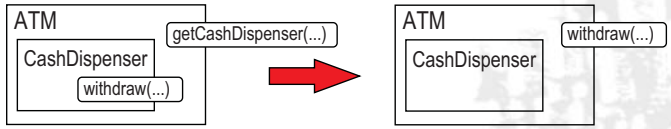
- Avoid dumb storage classes
- Keep related data and behaviour together
- Example (moving a point):



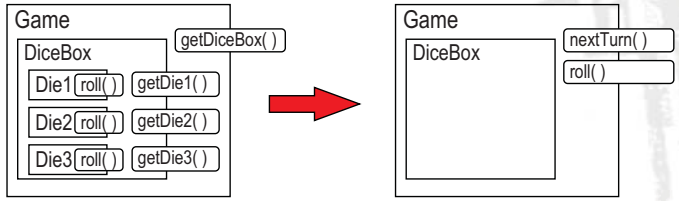
? Who uses the operations
? What are they doing with the data
? Why can't Point do it itself

Coping with the God Class Problem 2

- Hide local classes/ objects
- Examples:
  - Withdrawal using an ATM

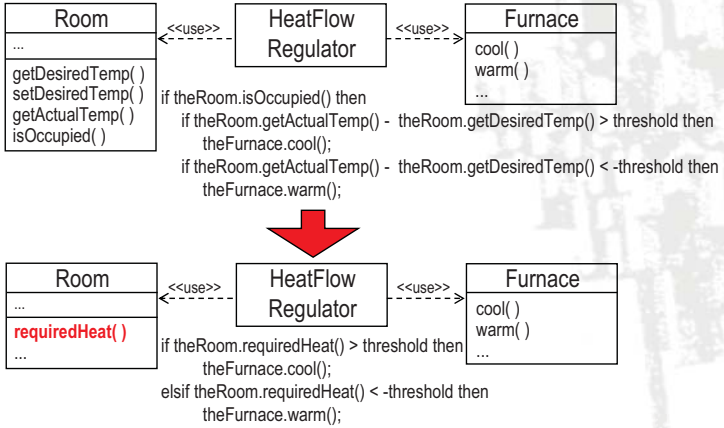


  - Rolling dice in a game



Coping with the God Class Problem 3

- Distribute System Intelligence
- Example (home heating system):



Coping with the God Class Problem 4

- Beware of controller classes
- Real-world examples:
  - VCR/ camera (data and behaviour is strictly separate)
    - Controller: The recorder/ player/ camera
    - Data: The tapes/ films
    - \+ Very flexible
    - – Complicated
    - – Expensive
  - Throw away camera has data and behaviour
    - \+ Cheap
    - \+ Easy to use
    - – Limited use
    - – Quality

Typical goals in the software world

Heuristics for Avoiding God Classes

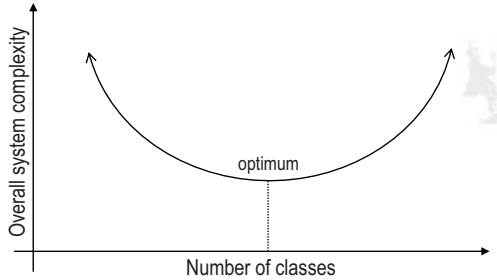
- Distribute horizontal system intelligence uniformly; the top-level classes should share the work uniformly
- Most of the methods of a class should use most of the data most of the time
- Spin-off non-related information into another class
- Beware of classes with many accessor methods in their public interfaces, especially if they do not have any behaviour
- Keep related data and behaviour in one place
- Be suspicious of any class whose name contains driver, (sub)system, manager, controller, ...

Design Heuristics and Patterns

- Introduction
- The "God" Class Problem
- The Proliferation of Classes Problem
- Conflicting Heuristics

The Proliferation of Classes Problem

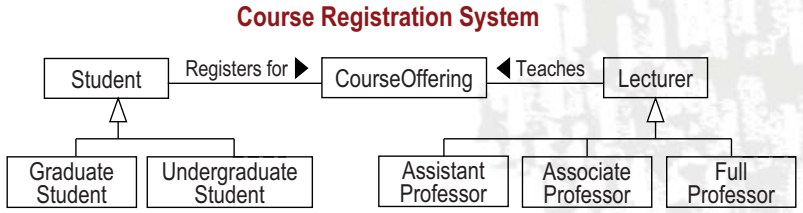
- "Spaghetti"- vs. "ravioli"-code discussion
- Too few classes lead to overly complex classes
- Too many classes increase overall complexity



➔ Avoid "unnecessary" classes

Coping with the Proliferation of Classes Problem 1

- Eliminate irrelevant classes without behaviour (i.e. containing only set-, get-, and simple print-operations)



- Attributes are probably sufficient here
- Sensors ("get-ors") or transducers ("set-ors") are typical exceptions to this rule

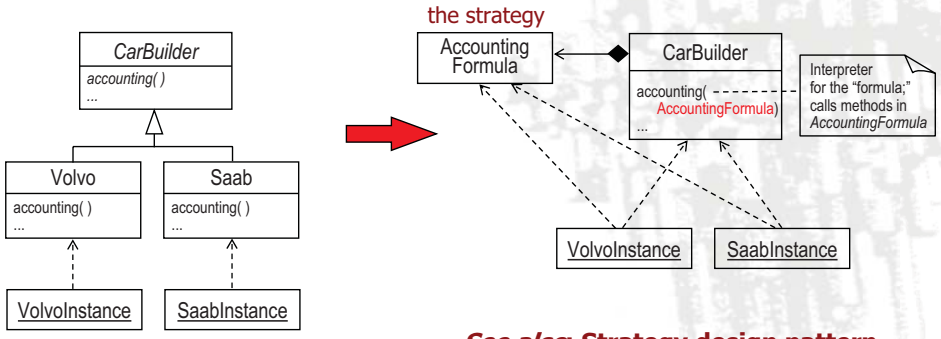
Coping with the Proliferation of Classes Problem 2

- Eliminate classes that lie outside the system
  - Examples:
    - Registrar in the Course Registration System
    - Customer in an ATM system
  - Tip: Actors are often outside the system
- Beware of irrelevant agent classes
  - Often useful during analysis, but irrelevant for design

- Do not turn an operation into a class
  - Behavioural god class!

Coping with the Proliferation of Classes Problem 3

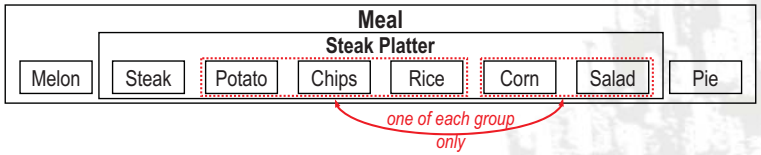
- Avoid subclassing when the subclasses have only a single instance
- Problem: Instances have different behaviour



***See also*: Strategy design pattern**

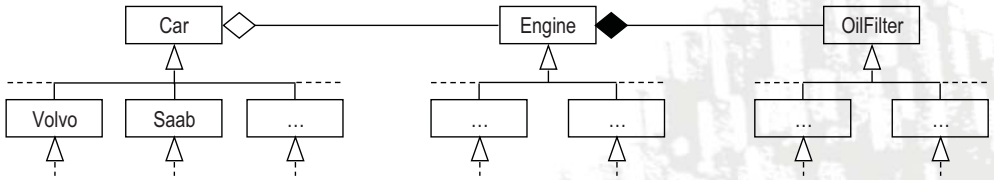
Coping with the Proliferation of Classes Problem 4

- Containment hierarchies and semantic constraints



- Usually in class definition(s) (constructors)
  - Potato and corn steak platter
  - Potato and salad steak platter
  - Chips and corn steak platter
  - ...
- Problem: Combinatorial explosion

Implementing Semantic Constraints—An Example



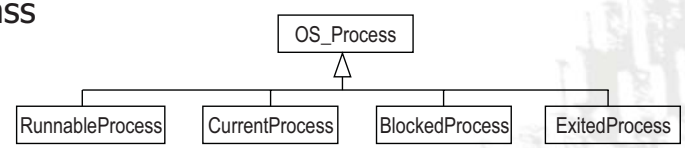
- General strategy
  - Build deep and narrow inheritance hierarchies and handle the constraints in the constructors (as far down as possible)
  - Allow the creation of "wrong" objects, but validate objects via methods
  - `StartEngine()` or `drive( )` methods check for correct combinations

Implementing Semantic Constraints (cont.)

- Information is volatile
  - Use a central third-party object
  - Example: Tables that match car models with suitable oil filters
- Information is stable
  - Decentralise among involved parties
  - Example: Each car maintains a list of allowed engines

Coping with the Proliferation of Classes Problem 5

- Do not use inheritance to model the dynamic semantics of a class



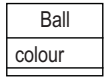
➔ Need to change types at runtime
➔ Information hiding?

- Are the subclasses really special types of operating system processes?

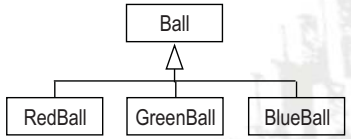
➔ No, they are the four states of an OS process
➔ Verify by means of statechart diagrams

Coping with the Proliferation of Classes Problem 5 (cont.)

- Which one is correct?



(a)

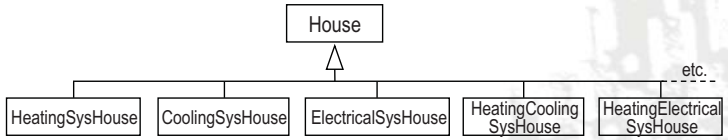


(b)

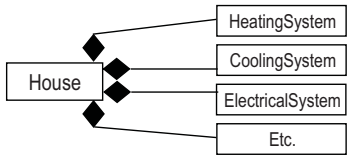
- If the value of an attribute affects the behaviour of the class, choose solution **(b)**
- Explicit case analysis on the value of an attribute is often an error
  - Transform into class hierarchy
  - Use dynamic binding
- Difference to previous example?

Coping with the Proliferation of Classes Problem 6

- Implementation of optional containment
- Using inheritance leads to combinatorial explosion:



➔ Do not confuse "has-a" with "is-a"



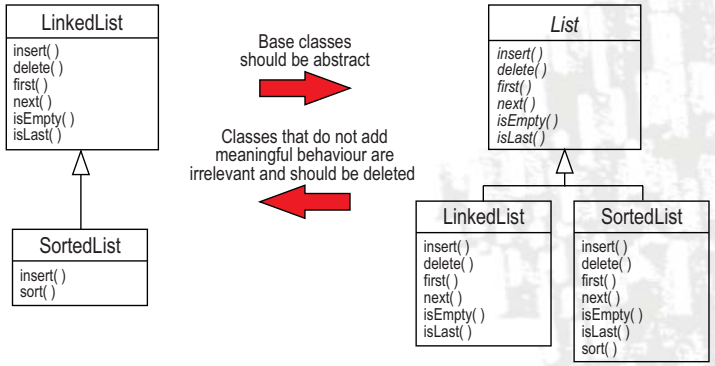
Design Heuristics and Patterns

- Introduction
- The "God" Class Problem
- The Proliferation of Classes Problem
- Conflicting Heuristics

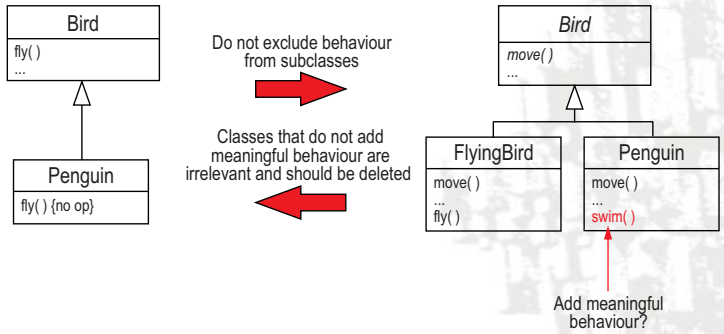
Conflicting Heuristics

- Heuristics are "rules-of-thumb," no laws
- The rules may give conflicting advise
- Example:
  - Base classes should be abstract
  - Classes that do not add meaningful behaviour are irrelevant and should be deleted
  - Do not exclude behaviour from subclasses

Conflicting Heuristics—Example 1



Conflicting Heuristics—Example 2



Design Guidelines and Patterns

- Introduction
- Design Principles
- Refactoring
- Design Heuristics
- Design Patterns
- References

Design Patterns

"Canned solutions to known problems." [A. Riel at OOPSLA´96]

Design patterns "are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations,and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue."

Design patterns "are based on practical solutions ..."

[GHJV 95], pp 3/4

Pattern Description Schemes

- GoF ([GHJV 95])
  - Name
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns
- Siemens ([BMRSS 96])
  - Name
  - Abstract
  - Also Known As
  - Example
  - Context
  - Problem
  - Solution
  - Structure
  - Dynamics
  - Implementation
  - Variants
  - Example Resolved
  - Known Uses
  - Consequences
  - See Also

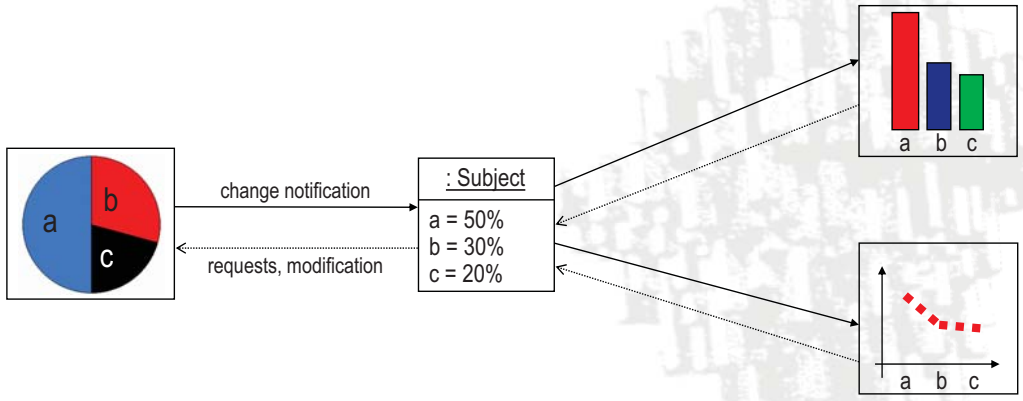
The GoF Patterns

	Object Creation	Class/ Object Composition	Object Interaction/ Responsibility Distribution
Class Level	Factory Method	Adapter (class)	Interpreter Template Method
Object Level	***Abstract Factory*** Builder Prototype ***Singleton***	Adapter (object) Bridge ***Composite*** Decorator ***Facade*** Flyweight ***Proxy***	Chain of Responsibility Command Iterator Mediator Memento ***Observer*** ***State*** Strategy Visitor

Good Patterns Resources

- The Hillside group
  - ***http://hillside.net***
- Pattern Stories Web
  - ***http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic***
- Non-software examples of (GoF) patterns
  - *Changes location from year to year (google it)*

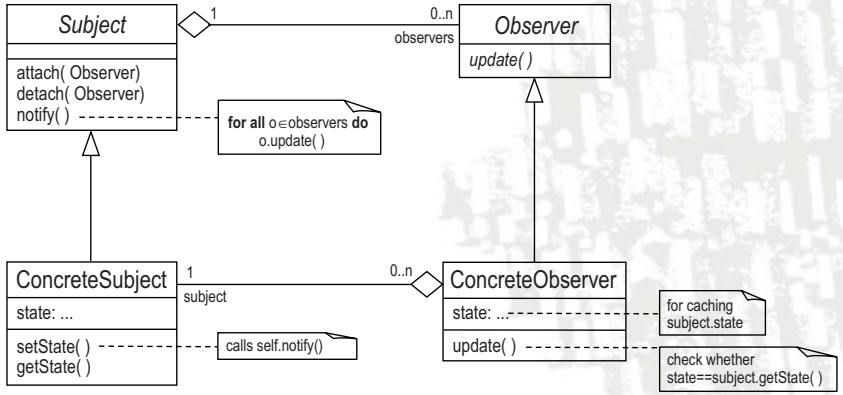
The Observer Pattern 1



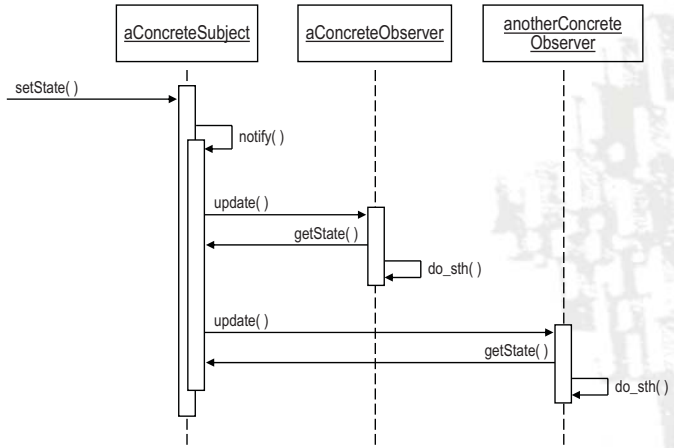
The Observer Pattern 2

- Notify-update mechanism (publish-subscribe)
- Originates from Smalltalk´s MVC
- Used in almost all GUI libraries/toolkits
  - Usually only two explicit components: Model and View&Controller
  - JFC examples:
    - ListModel, ListDataListener, JList
    - TableModel, TableModelListener, JTable
    - …
- Advantages
  - Uncouples dependent components
  - Increases flexibility
  - Increases reusability

The Observer Pattern 3



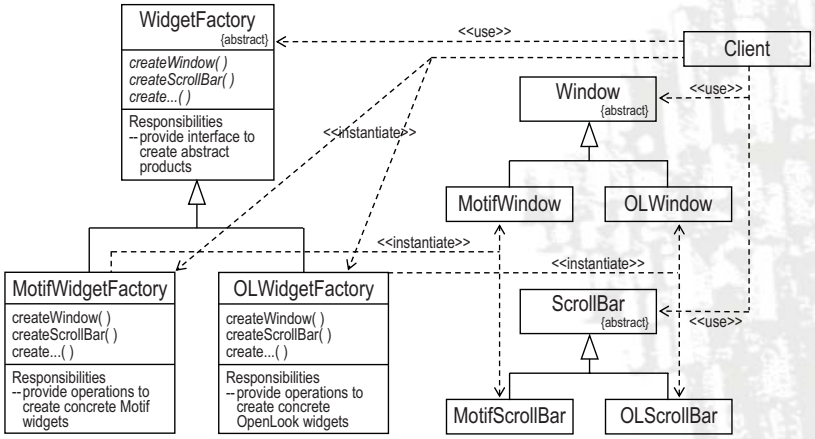
The Observer Pattern 4



The Abstract Factory Pattern 1

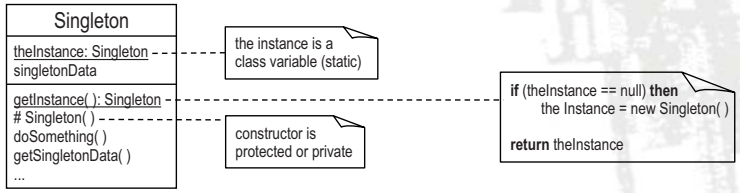
- Interface for creating interdependent objects without specifying their concrete classes
- Clients use only abstract classes to handle products
- Selection of the correct concrete classes is determined by the concrete factory
- Widely used in the JDK (look-and-feel handling, BorderFactory etc.)
- Advantages
  - Hides concrete classes and their interdependencies
  - Simplifies consistency management

The Abstract Factory Pattern 2



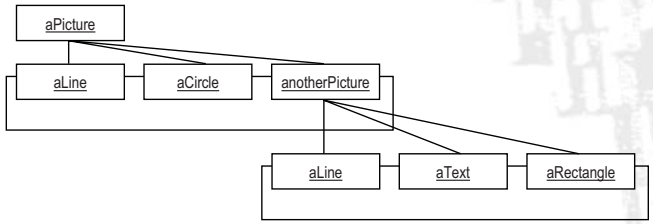
The Singleton Pattern

- Class with a single, globally accessible instance
- Creation can be on demand
- Widely used in the JDK
- Advantages
  - Reduced name space
  - Simple realisation of sharing
  - OOPS! Much more flexible than class methods only

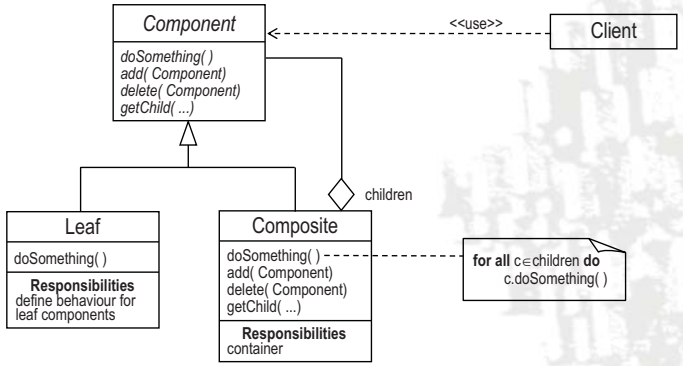


The Composite Pattern 1

- To handle composed objects (containers) and their components in an uniform way
- Common problem in document composition and graphics
- Advantages
  - Less need for case statements
  - Easy to add new kinds of components

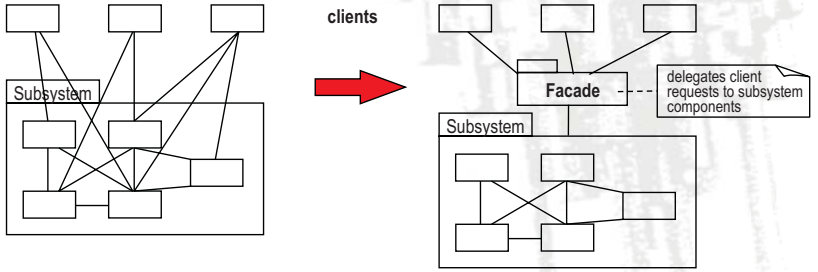


The Composite Pattern 2



The Facade Pattern

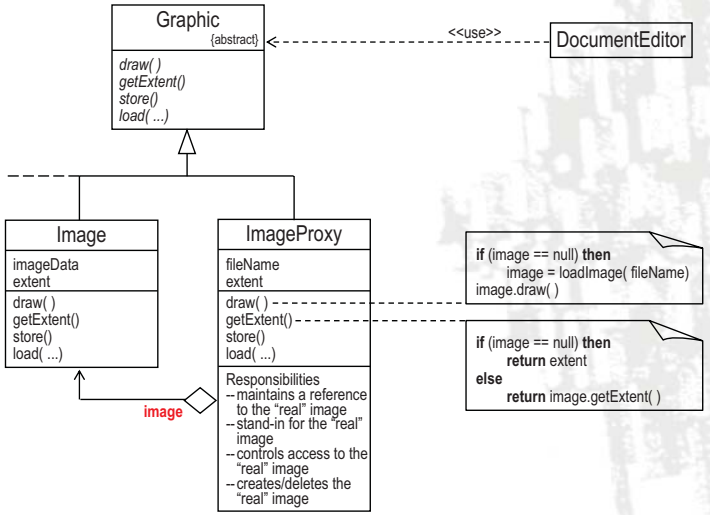
- Defines a unified, high(er)-level interface to a (complex) subsystem
- Advantages
  - Hides internal subsystem structure
  - Simplifies subsystem use
  - Weakens coupling between clients and subsystem



The Proxy Pattern 1

- Provide an approximation that can "stand-in" for another object
  - Remote proxy to provide a local representative for a remote object
  - Virtual proxy to create expensive objects on demand
  - Protection proxy to control access to the original object
  - Smart reference to give "added value" to bare pointers
- Widely used
- Advantages
  - Reduce handling costs for heavy weight objects
  - Hide distribution details
  - Access protection

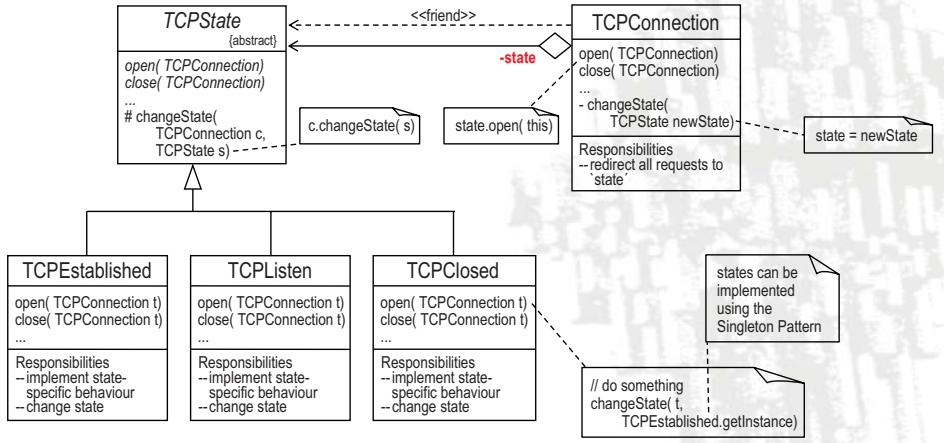
The Proxy Pattern 2



The State Pattern 1

- To allow objects to change their behaviour depending on their (internal) state
- "Simulate" dynamic type changes
- Advantages
  - Localises and partitions state-dependent behaviour
  - Makes state transitions explicit
  - State objects are explicit and can be shared

The State Pattern 2



Summary

	Class Libraries	Frameworks	Heuristics	Patterns
Abstraction	concrete classes	interrelated classes (concrete and abstract)	rules	problem-solution descriptions
Programming Language	specific	specific	independent	independent
(Re)use What	classes	classes and infra-structure (⇨ design)	experiences	experiences
How	as is	subclassing	"rules-of-thumb"	"blueprints"
When	coding	design/coding	design	analysis/design
Adaptability	low	extensible	generic	generic

Contents

References

[BMRSS 96] F. Buschmann, R. Meunier, H. Rohnert, Peter Sommerlad, M. Stahl, *A System of Patterns*, Wiley, 1996.

[Fowler 99] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[GHJV 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Krist 94] G. Kristensen, *Object Orientation: The KISS® Method*, Addison-Wesley, 1994.

[LiHo 89] K.J. Lieberherr, I.M. Holland, Assuring Good Style for Object-Oriented Programs, *IEEE Software* **6**(5), Sep 89, 38-48.

[Martin 02] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2002.

[Pree 95] W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.

[Riel 96] A. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.