University of Regensburg

Department of Physics

Software Design Patterns

Introduction to modern software architecture

Florian Rappl

Contents

Chapter 1

General

1.1 Structure

- We'll have around 12 lectures (90 min. each)
- There will be around 6 exercises (every 2nd week, starting in November)
- The final exam will be given in form of a project
- Here principles from the lecture should be applied
- This lecture represents a complete module
- Classification is given by M 32

1.2 Exercises

- Exercises are mostly practical
- Nevertheless some (mostly in the beginning) of the exercises are of theoretical nature
- Practical exercises will be solved on the computer
- Theoretical exercises should be solved on paper
- The exercises will take place on some CIP pool

1.3 Exercises appointment?

- We still need a point in time and space
- The time coordinate will have an influence on the possible choices for the location
- Hence an agreement has to be found
- Possible to do a short Doodle-like survey to have a majority vote

1.4 Are you right here?

- The next couple of slides will discuss what
  - will *not* be discussed
  - will *be* presented in the lecture
- It is worth noting that programming knowledge *is* actually required
- The language constraint is soft with C++, C# or Java being preferred
- OOP (**what's that?!**) knowledge is not required, but basic understanding is helpful (what is a *class*, what is an *object*, ...)

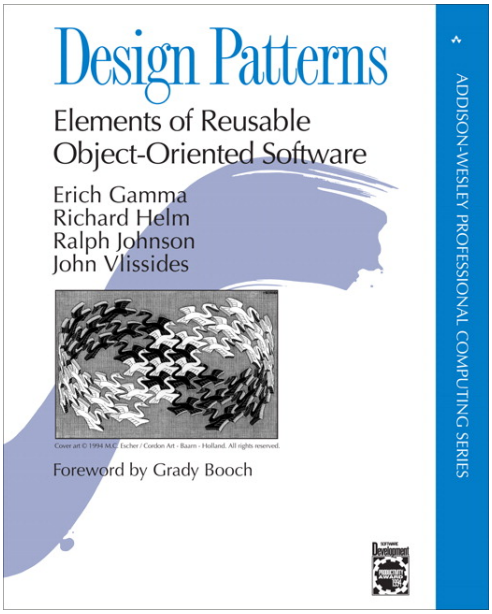
1.5 It's not about ...

- A pure lecture on UML (however, this is an important part of the lecture)
- Programming language dependent patterns
- Scripting and dynamic languages
- Learning how to become a programming *rockstar*
- Features of some languages (attributes, templates, generics, LINQ, dynamics, ...)

1.6 Covered

- UML
- OOP (introduction of C++, C#, Java)
- Design Patterns (creational, structural, behavioral, concurrency and representational)
- Separation of Concerns
- SOLID
- Clean Code
- Refactoring

1.7 A book (covered, not required)



1.8 And finally...

- The lecture should be dynamic enough to cover topics interesting for **you**
- If something is unclear, not detailed enough or too theoretical then just interrupt
- The idea is to adapt the content to *your needs*
- Currently planned is about one third theory like UML and two thirds patterns and practices

1.9 Software architecture

- The title of the lecture is "Software Design Patterns", but in fact we are looking at software engineering in general
- This consists of the following parts:
  - Software architecture describes how overall projects are going to be structured
  - Design patterns allow us to reuse existing solutions and experiences
  - Principles tell us how to achieve a maintainable, extensible yet lightweight coding style

1.10 References

- Official lecture webpage (http://www.florian-rappl.de/Lectures/Page/181/sdplectureoverview)
- IT lectures in the winter term 2013/14 (http://www.physik.uni-regensburg.de/studium/edverg/termine_ws1314)
- Description of the lecture (http://www.physik.uni-regensburg.de/studium/edverg/patterns/)
- HIS-LSF (http://lsf.uni-regensburg.de/qisserver/rds?state=wtree&search=1&trex=step&root120132=11136%7C8815%7C9213&P.vx=mittel)
- Formalities as stated from the RZ (http://www.uni-regensburg.de/rechenzentrum/lehre-lernen/it-ausbildung/aufbau/)

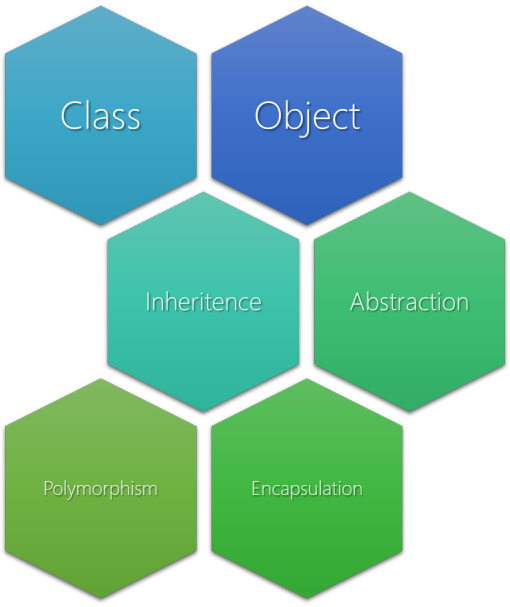
Chapter 2

Basics

2.1 What is OOP?

- OOP is a different way of thinking in programming
- Instead of having only functions that modify data we have objects that carry data
- These objects also have associated functions (then called methods)
- Advantage: Clear separation possible
- Such objects are usually so called instances of classes
- Objects are described by the data they carry (which are called fields)

2.2 OOP Overview



2.3 Structure Vs Class

- Creating own data types is mostly based on either structures or classes
- Structures should be used for immutable, primitive, i.e. small and atomic, types
- Classes should be seen as complex, i.e. compound, types that are build from other types
- Some languages impose restrictions and behavior depending on which kind is chosen (mostly managed languages)
- In this lecture we will exclusively use classes

2.4 Features

- Encapsulation enforces modularity
- Inheritance creates groups of similar objects and makes classes re-usable
- Polymorphism lets specializations implement their own behavior
- Decoupling makes code highly testable, exchangeable and general

2.5 OOP languages

- Simula (1967) initiated the idea
- OOP became popular with Smalltalk (mostly due to GUI)
- In the 80s Ada and Prolog followed
- Also C++ and Objective-C were introduced
- With Java the ideas of IL, managed code and OOP merged
- Finally C# took the ideas of Java even further

2.6 Language distinctions

- Some languages are *pure*: Eiffel, Scala, Smalltalk, Self, ...
- Others are focused on OOP (they *still have* procedural elements): Delphi, C++, C#, Java, Python, JavaScript, ...
- Some are derived from procedural ancestors: Pascal, VB (from Basic), Fortran, Perl, PHP, COBOL, ...
- There are also *object-based* languages, i.e. they support data types but not all OO features (like Modula-2)
- We will focus on C++, C# and Java

2.7 Hello World

```
using System;

public class Program
{
    public static void Main(String[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

2.8 Basics of C#

- class creates a class
- static marks methods as being instance independent
- using imports types from namespaces
- public sets members as being visible outside of the class
- private members are accessible from the outside
- Primitive types like in C (int, char, double, ...)

2.9 Basics of Java

- Overall quite similar to C#
- Inheritance is done by a keyword (extends) and not an operator (:)
- Differentiates in inheritance kind by a keyword (implements)
- Types are imported with the import keyword
- The protected modifier is like private, but lets derived classes access the members

2.10 Basics of C++

- We have to take care of the memory
- Classes should always be created with new
- Accessing member variables of objects is then done by ->, as opposed to the . in other languages
- Freeing objects can be achieved by calling delete (beware of arrays!)
- Modifiers do not have to be set explicitly, but for blocks
- Header files have to be included as well

2.11 Constructors

- These are functions that are called implicitly after allocating memory
- Special form: No return value and same name as the class
- They can be used for doing initialization or checking on conditions
- They could throw exceptions
- We can specify arguments to impose dependencies
- Constructors also have a modifier, i.e. they could be restricted to be accessed from derived class constructors or only within the same class
- This can result in a very handy pattern: the so called **Singleton** pattern

2.12 Polymorphism

- Classes allow us to implement a method and re-implement the same method in a derived class
- This feature is called polymorphism
- The idea is that even though one talks to a more general kind of type (not knowing the specialized one) one accesses the special method
- The method is selected by a method selection algorithm
- This is either done at compile-time (special overload, C++) or at run-time (virtual method dispatch over a function table)

2.13 Polymorphism example

```
public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Base draw");
    }
}
class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}
class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}
```

2.14 Language implementation

- In Java every method is virtual, except those which are marked final (cannot be overridden or hidden)
- C++ does not require placing an override keyword
- The @Override annotation in Java prevents typing mistakes (it is not required, but strongly recommended)
- In C# the override is required, even though the method has been marked as virtual

2.15 Abstract

- Previously we have seen a way to re-implement methods
- Additionally we can enforce implementation
- An abstract method is such a method that is required to be re-implemented
- Abstract classes are classes that can never be instantiated
- These two concepts build the foundation of many patterns

2.16 Abstract example

```
public abstract class Ball
{
    public abstract int Hit(int speed);
}
public class BaseBall : Ball
{
    public override int Hit(int batSpeed)
    {
        //Implementation
    }
}
```

2.17 Abstract usage

- C++ does not have special keywords
- Any class having abstract methods is abstract
- Abstract classes have to be marked as such in C# or Java
- It is not possible to create instances of abstract classes
- Abstract classes are useful to group something which has at least one method, which requires different implementation

2.18 Interfaces

- A step further than abstract classes are interfaces
- An interface is like an abstract class without any code
- It could be seen as a contract (sets up what kind of methods should be available)
- In modern software development interfaces are crucial
- Goal: Reduce dependencies and decouple classes

2.19 Interfaces in practice

```
interface IShooter
{
    int Ammo { get; set; }
    void ShootAt(int x, int y);
}
class Player : IShooter
{
    /* Implementations */
}
```

2.20 Worth noting ...

- C# offers special constructs called properties (otherwise we prefer to use the so-called bean convention)
- In C++ interface-like types can be created by abstract classes that contain only abstract methods marked as public
- Java requires a different keyword for implementing interfaces (however, for us this is educational)
- In C# and Java methods in interfaces have implicitly a public modifier

2.21 Comparison

C#	Java	C++
Managed	Managed	Native
Pointers possible	No pointers	Pointers required
High OO	High OO	Low OO
With Mono	Cross platform	With Qt
C# 5	Java 7	C++11
Lambdas	Proxy	Lambdas
Events	Custom	Custom

2.22 The diamond problem

- C++ allows multiple inheritance, plus merging again
- Here we can build a diamond structure with a base class A, two subclasses B and C and a class D, that inherits from B and C
- Problem: We now have the methods have A duplicated
- Also the data of A is duplicated
- Solution: Instead of doing a normal inheritance, do a virtual one

2.23 Diamond hierarchy

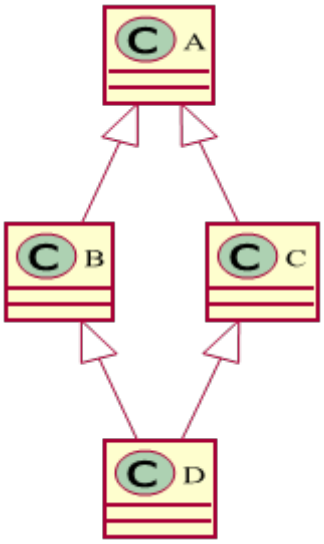


Figure 2.1: Diamond problem.

2.24 A look at the creators

C#

Hejlsberg
Java

Gosling
C++

Stroustrup

2.25 Other languages

- Objective-C is an OO language that is empowering iOS and OSX applications
- Vala is using a lot of C# and D concepts to create native code

- Eiffel is build upon design by contract and is enforces OO principles
- JavaScript is a fully OO scripting languages, however, it is prototype-based and not class-based
- Visual Basic is a popular extension of Basic with OO in mind
- Functional OO languages like F#, Scala and others mix concepts

2.26 Design patterns

- A general reusable solution to a commonly occurring problem
- The context for the problem (and solution) is software
- Since it is a pattern it can be transformed to apply to our problem
- They usually follow best practices, even though the implementation might not
- Languages might enhance using these patterns
- Sometimes languages already provide integrations, which should be preferred

2.27 References

- Oracle: Official Java tutorials (`http://docs.oracle.com/javase/tutorial/`)
- German Java tutorial (`http://www.java-tutorial.org/`)
- MSDN: Official C# tutorials (`http://msdn.microsoft.com/en-us/library/aa288436.aspx`)
- An advanced introduction to C# (`http://tech.pro/tutorial/1172/an-advanced-intro`
- C++ language tutorial (`http://www.cplusplus.com/doc/tutorial/`)
- Learn C++ (`http://www.learncpp.com/`)

2.28 Literature

- Drayton, Peter; Albahari, Ben; Neward, Ted (2002). *C# Language Pocket Reference.*
- Gosling, James; Joy, Bill; Steele, Guy L., Jr.; Bracha, Gilad (2005). *The Java Language Specification.*
- Alexandrescu, Andrei; Sutter, Herb (2004). *C++ Design and Coding Standards: Rules and Guidelines for Writing Programs.*
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object Oriented Software.*

Chapter 3

Software Life Cycle

3.1 Software development processes

- A software development life-cycle (short SDLC) structures development
- This is often called software development process
- We can categorize various different approaches by their attitude towards:
  - Planning
  - Implementation, testing and documenting
  - Deployment and maintenance
- Most of this is only of academic interest

3.2 Planning

- A crucial phase as requirements are set
- Also a draft for the software specification is created
- Better planning enables a better (faster, bug-free and feature-rich) development, but requires a lot experience
- Sometimes planning can also result in contracts and legal documents (especially for individual software)
- The requirement analysis can also contain prototype creation

3.3 Implementation

- Actually writing code with logic is called implementation
- This phase could be broken in various sub-phases (prototype, API, realization, ...)
- Usually the whole process should divided into little tasks

- Developers take care of these little tasks
- Engineers keep in mind the big picture

3.4 Testing

- Testing is important to ensure quality
- Goal: Minimizing bugs, maximizing usability and stability
- There are multiple kind of tests: automatic, manual, usability, performance, ...
- Some of them are contradictions (there cannot be automatic usability tests), some of them should be combined (like automatic code unit tests)
- There are movements towards dictating the implementation by automatic tests (see TDD chapter)

3.5 Documentation

- Documentation is important due to various reasons
  - Manuals for users (*top level*)
  - Manuals for other developers (*low level*)
  - Legal documents (*fulfillment*)
  - Finding bugs and improving code
- Documentation should not only be done in documents but also in comments in the code
- Many tools can take comments and create documents

3.6 Deployment

- Finally a software product can be released
- More than just releasing a binary might be required:
  - Installation
  - Documentation
  - Customization
  - Testing and evaluating
- Training and support might demanded as well

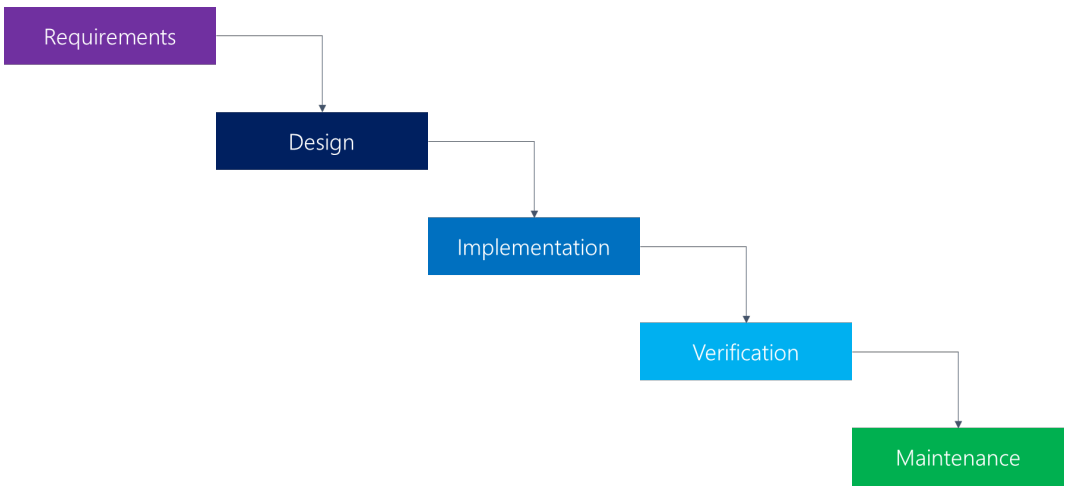
3.7 Maintenance

- Software is never finished
- Updates may be needed due to include new features or fix bugs
- Here we are usually out of the requirement analysis scope
- This is where excellent software architecture shines
- Making software maintainable is the hardest task

3.8 Existing models

- **Waterfall** model
- Spiral model
- **Iterative** and incremental development
- **Agile** development
- Rapid application development

3.9 Waterfall model

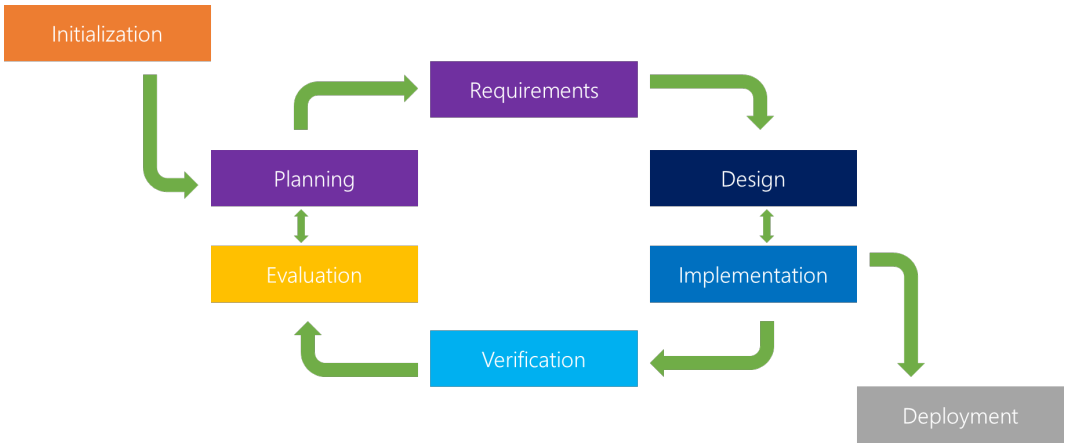


3.10 Predictive and adaptive planning

- Desire for predictability in software development
- Predictive planning uses two stages:
  - Coming up with a plans (difficult to predict)
  - Executing the plans (predictable)
- However, majority suffers from a requirement churn

- Let's face it: the requirement churn cannot be avoided
- Now we are free to do adaptive planning, i.e. trying to deliver the best software and react to changes

3.11 Iterative model



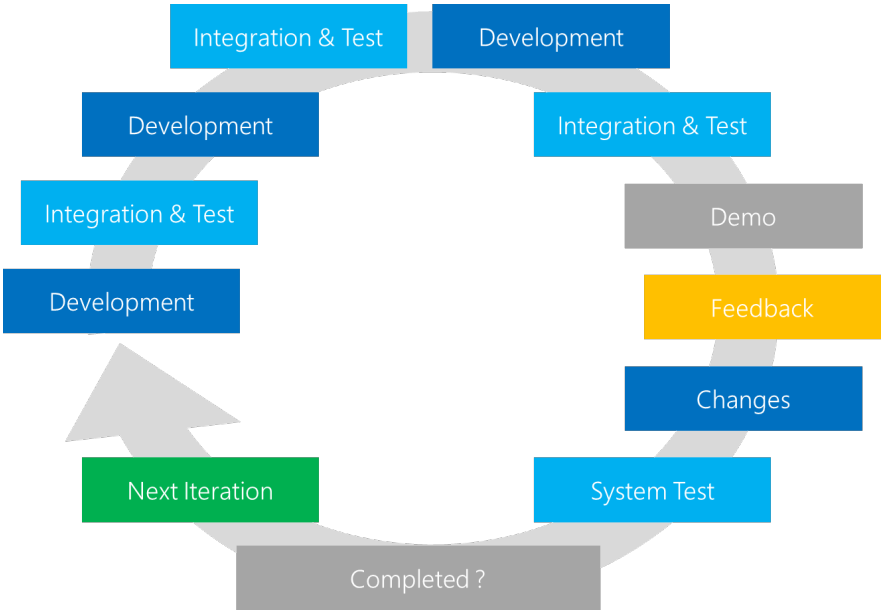
3.12 Waterfall Vs Iterative

- Both break a project into smaller chunks
- Waterfall does it based on *activity*
- Iterative identifies subsets of *functionality*
- No one has ever followed one of those purely
- There will always be impurities leaking into the process
- Iterative development can also appear as incremental, spiral, evolutionary etc.

3.13 Staged delivery

- Pure waterfall has never been used and waterfall is generally disliked
- One can have an hybrid approach that takes a subset of waterfall and the iterative process
- Analysis and high-level design are usually done first
- Coding and testing phases are then executed in an iterative manner
- Main advantage with iterative coding: Better indication of problems
- Consider using *time boxing* to force iterations to be a fixed length of time

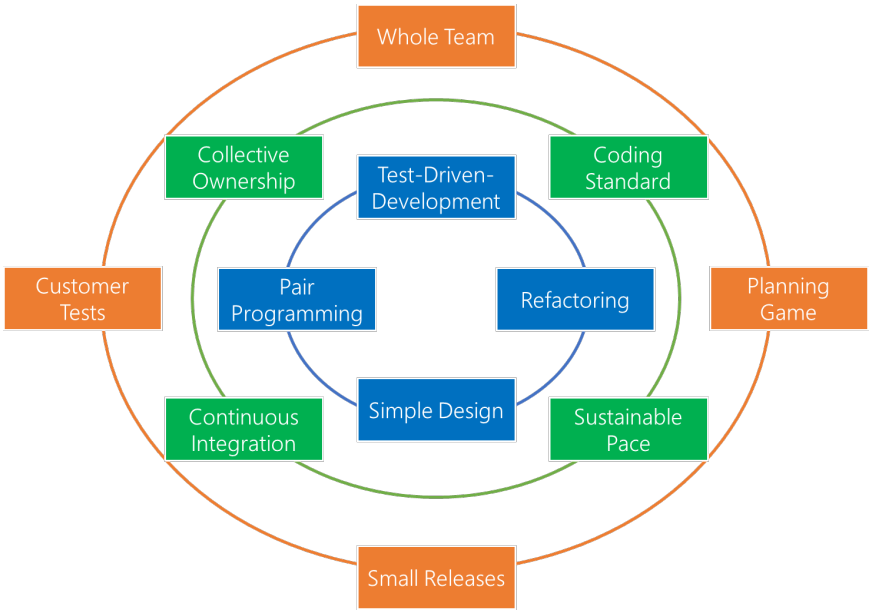
3.14 Agile software development



3.15 Agile processes

- Agile is an umbrella term that covers many processes
- Examples: Extreme Programming, Scrum, Feature Driven Development
- Very adaptive in their nature
- Quality of people and their interaction results in quality of code
- Agile methods tend to use short, time-boxed iterations (a month or less)
- Usually agile is considered lightweight

3.16 Extreme Programming



3.17 Rational Unified Process

- A process framework
- Provides vocabulary and structure for discussing processes
- A development case needs to be selected
- RUP is essentially an iterative process and has four phases:
  1. Inception (initial evaluation)
  2. Elaboration (identifying requirements)
  3. Construction (implementation)
  4. Transition (late-stage activities)

3.18 Techniques

- Developing software efficiently is not only a matter of the planned life cycle, but also of using techniques like
  - Version controlling (e.g. git)
  - Automated regression tests (e.g. JUnit)
  - Refactoring (e.g. VS)
  - Continuous integration (e.g. TFS)
- We will have a look at two of those techniques later

3.19 References

- What is Agile? (http://agileinsights.wordpress.com/2013/08/26/what-is-agile/)
- What is SCRUM? (http://www.codeproject.com/Articles/4798/What-is-SCRUM)
- Staged delivery (http://lauyin.com/NPD/model/staged.html)
- Software development process (http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Software_development_process.html)
- Rational Unified Process (http://www.projects.staffs.ac.uk/suniwe/project/projectapproach.html)
- Why do companies use agile development? (http://www.techwench.com/why-do-companies-use-agile-development/)

3.20 Literature

- Fowler, Martin (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.).*
- McConnell, Steve (1996). *Rapid development: taming wild software schedules.*
- Kent, Beck (2000). *Extreme Programming.*
- Highsmith, James (1999). *Adaptive Software Development.*

Chapter 4

UML

4.1 What is the UML?

- UML means *Unified Modeling Language*
- It is a family of graphical notations with a single meta model
- Purpose: Describe and design software systems
- In reality it can be used for a lot of things
- UML is an open standard that is controlled by the OMG (short for *Object Management Group*)
- It is used for doing MDA (Model Driven Architecture)

4.2 Official logo

- It was released in 1997, with a major update 2005 (**v2.0**)
- The current version of the UML is 2.4.1 (published in August 2011)

4.3 UML modes

- Three modes: *sketch*, *blueprint* and *programming language*
- UML as a sketch is the most popular
- Reason: Organization, selectivity, discussion
- Blueprint diagrams are about completeness
- They have to be very detailed, such that programmers only need to follow them
- It is possible that UML represents source code, which can be compiled and executed

4.4 UML in practice

- Four great ways to utilize UML:
  1. Understanding legacy code (e.g. class)
  2. Documentation (e.g. state)
  3. Design (e.g. package)
  4. Requirement analysis (e.g. use-case)
- UML is a great for communication with non-software people
- Important here: Keep the notation at a minimum

4.5 Diagrams

- UML 2 has 13 different diagram types
- The standard indicates that certain elements are typically drawn on certain diagram types
- But: diagrams are not the central part
- And one can legally use elements from one diagram in other diagrams
- Important for us: class, object, activity, use-case and sequence
- Others: communication, component, composite, deployment, interaction, package, state, timing

4.6 Classification of diagrams

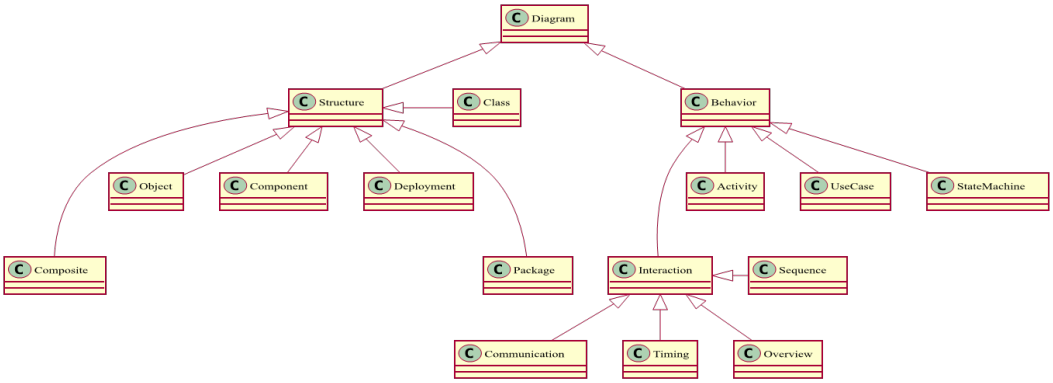


Figure 4.1: Classification of diagrams.

4.7 Legal UML

- Legal UML is how it is set by the standard
- The problem is the complexity of the specification
- But: UML is not prescriptive like programming languages
- Instead the UML is actually a descriptive language
- Hence the UML is a mix of independent conventions and official notations
- This results in the UML standard being a guide, not a burden

4.8 A sample class diagram

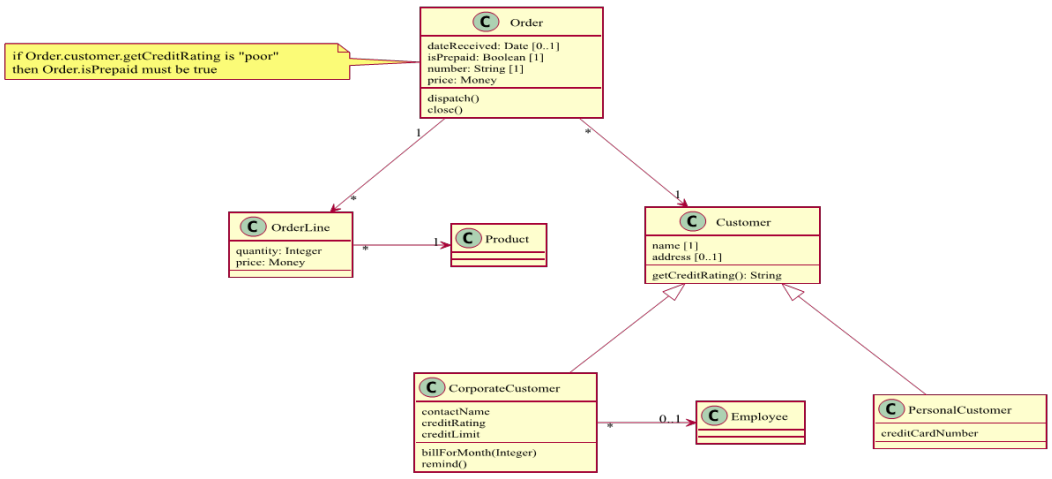


Figure 4.2: A sample class diagram.

4.9 Class diagrams

- Here boxes represent classes
- The name is on top, usually marked bold
- Two optional sections: Attributes and operations, with syntax:*visibility* name : type *[multiplicity] = default property*
- Attributes could be fields or properties (this implies get / set methods)
- Visibility: + (public), - (private), # (protected) or (package)
- A name is required, its type is sometimes left out
- Default value and property (like readonly, frozen, ...) are optional

4.10 Relationships

- Associations are drawn by a solid line
- Such associations can either be uni- (from source to target) or bidirectional (using a simple arrow)
- Generalization: A solid line that ends with a closed arrow (looks like a triangle) at the general class
- Notes or remarks are attached with a dashed line (no arrow required)
- Dependencies are represented by dashed lines

4.11 Aggregation Vs Composition

- Sometimes we want a special kind of association, either:
  1. Aggregation (opaque rhombus,)
  2. Composition (filled rhombus,)
- (1) is weak and specifies an optional association, like ***consisting of***
- (2) is strong and specified a requirement: like ***is part of***
- Composition also implies a no-sharing rule, i.e. a single class cannot be part of two classes directly, but only of one

4.12 Aggregation and Composition

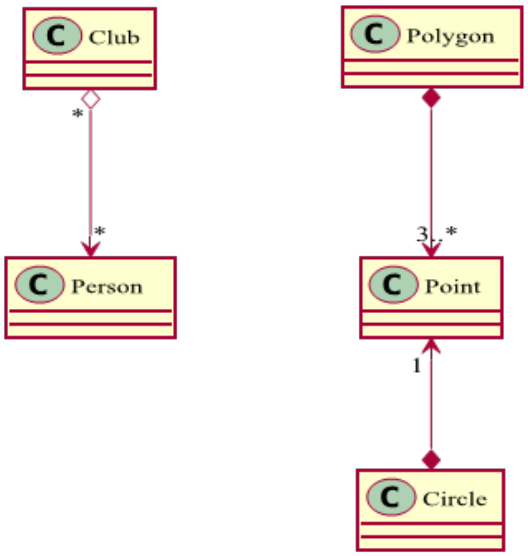


Figure 4.3: Aggregation and Composition.

4.13 Keywords and more

- We an annotate classes with keywords like *«interface»* or *«abstract»*
- Additionally we can mark static attributes and operations by underlining their definition or using the underscore _ symbol
- Difference between operation and method: A method is the implementation (body), while an operation represents just the name
- Some keywords are usually abbreviated (like *A* for abstract or *I* for interface)

4.14 Generalization

- Expression of is a relationship: Substitutability
- Maps directly to inheritance in most OOP languages
- Subclass/derived class is a generalization of superclass/base class
- Highest semantically defined relationship
- The purpose of generalization (inheritance) is to solve design problems
- Don't use generalization if there is no design problem

4.15 Another class diagram

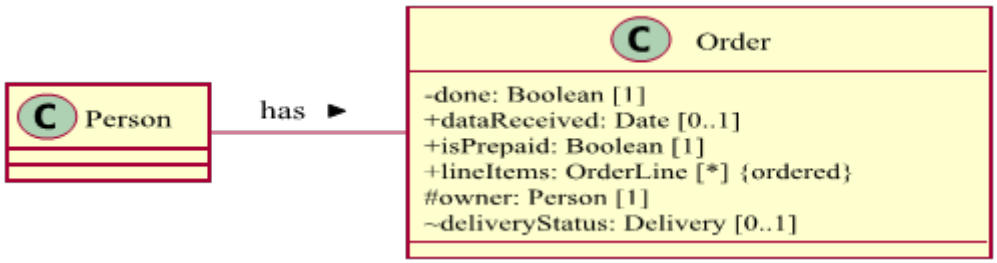


Figure 4.4: Visibility and bidirectional assocation.

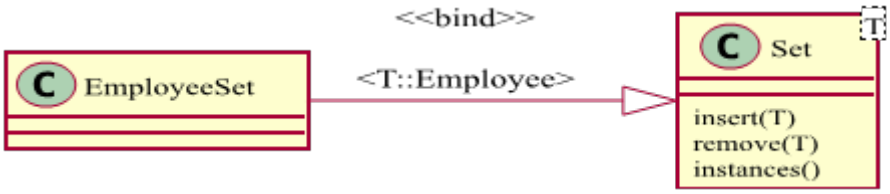


Figure 4.5: Template arguments.

4.16 Interfaces

- All operations are public, and no operation has a method body
- Indicated as keyword *«interface»*, or with a label *interface*, or abbreviated label *I*
- In this case, inheritance means implementation
- We can also have a dependency relationship with an interface
- Additionally ball-and-socket notation very common
- Ball - class provides interface (labeled), Socket - class requires interface

4.17 Example

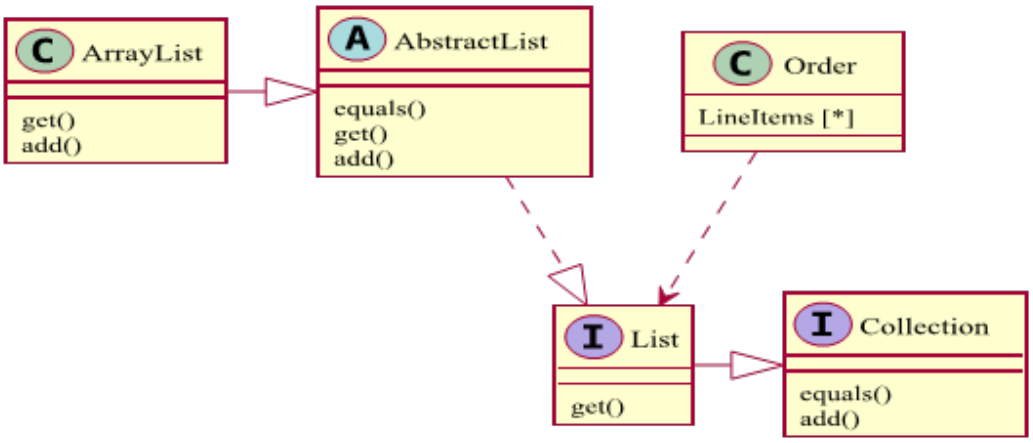


Figure 4.6: Interfaces with the UML.

4.18 Derived properties

- Derived properties are attributes
- They start with a forward slash /
- In general they represent computed values, i.e. a combination of other attributes (usually there are multiple equal choices)
- Another name is computed value
- Very useful to remind people of the underlying constraints

4.19 Two examples

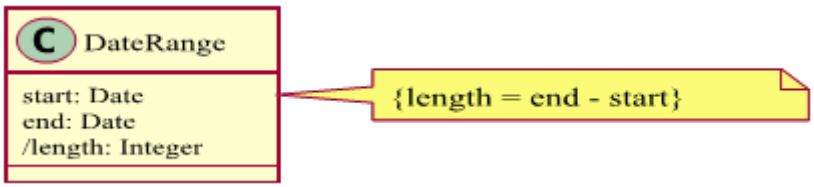


Figure 4.7: Derived attribute.

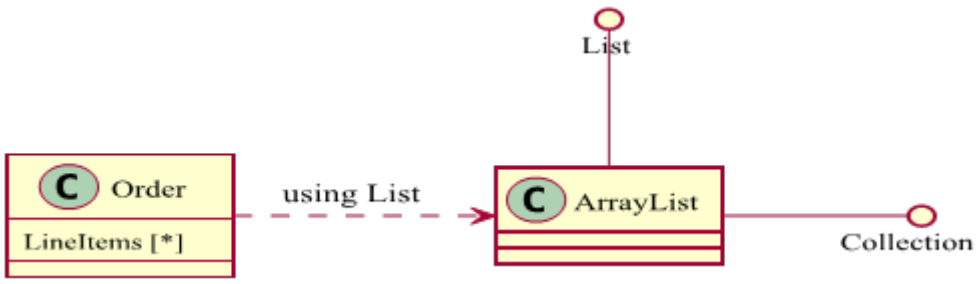


Figure 4.8: Ball-and-socket notation.

4.20 Even more notation

- *«struct»* for symbolizing value types (should be immutable)
- *«enumeration»* for creating an enumeration

Figure 4.9: Enumerations with the UML.

- Additionally templates are possible by placing a dashed box in the upper right corner (containing the template parameter(s))
- Marking an active class is possible by using two vertical lines

4.21 Object diagrams

- Closely related to class diagrams
- Shows instances at a given time frame
- Sometimes therefore called instance diagram
- Usage e.g. showing configuration of objects
- Names are not in **bold**, but underlined like Instance : Type
- Values do not need types and multiplicities (also no methods in general)
- Values are now mandatory like *location* = *"Boston"*, or *state* = *false*

4.22 An object diagram

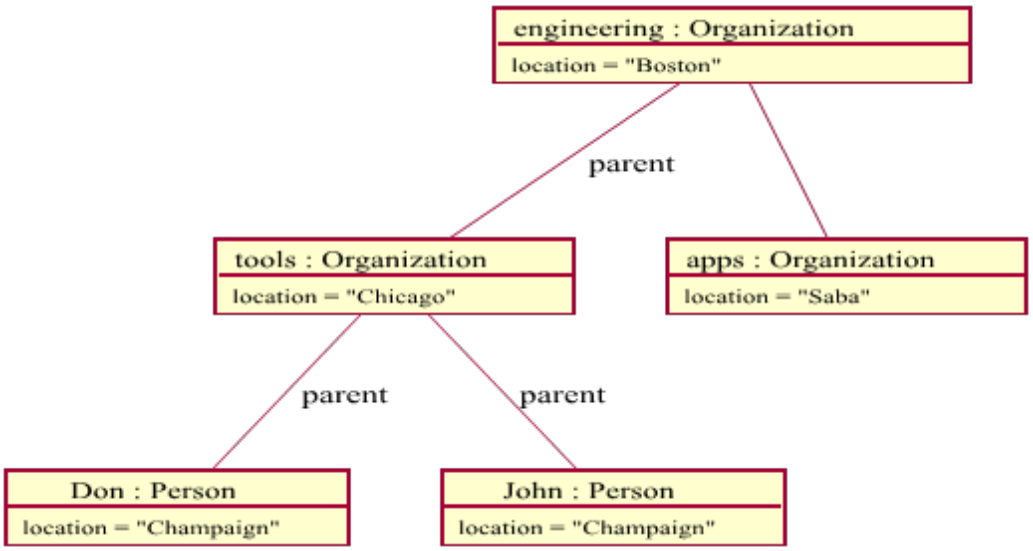


Figure 4.10: Object sample.

4.23 Activity diagrams

- Technique to describe procedural logic, business process and work flow
- Quite similar to flowcharts, *but* they support parallel behavior
- One new symbol for **fork** (e.g. 1 in, 2 out) and **join** (e.g. 2 in, 1 out)
- Initial node is a filled circle
- Actions are placed in rounded rectangles
- Decisions are symbolized by rhombi
- Activity final is represented by a bullseye

4.24 Example

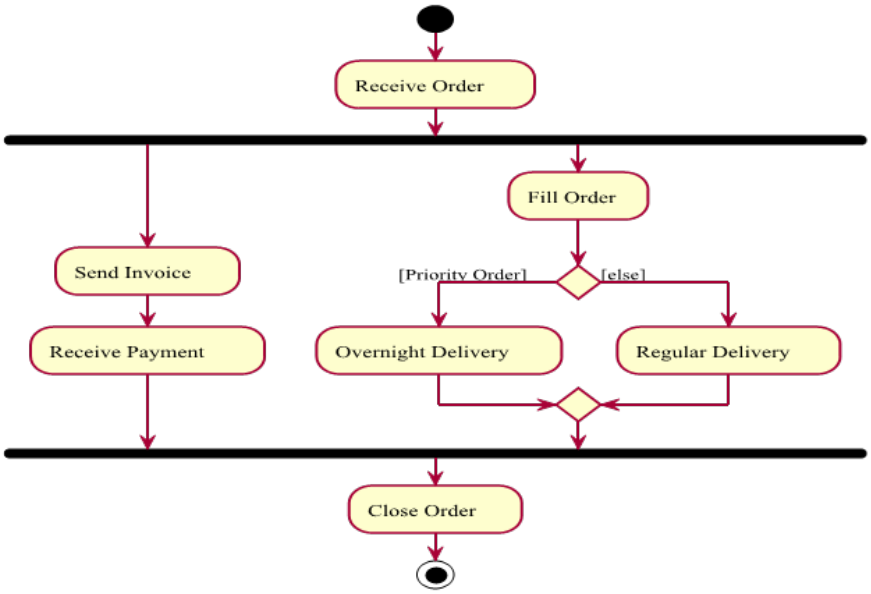


Figure 4.11: Activity sample.

4.25 Technical remarks

- Here nodes are called actions, i.e. we have a sequence of actions
- A **decision** is called a *branch*
- Square brackets contain *guards*, where else is a special guard that the flow should be used if no other guard applies
- A merge has to come after a decision (marked by a rhombus), with several incoming and one outgoing flow
- Additionally one might want to use partitions or even events called *signals* (which will not be introduced here)

4.26 Use Cases

- Capture the functional requirements of a system
- An **actor** is a central node type in such a diagram (sometimes called *role*)
- Common information could be added:
  - A pre-condition how the system should look like
  - A guarantee what the outcome is going to be
  - A trigger when to start the use-case
- Also differentiate between *fish-level* (only included in higher levels), *sea-level* (standard) and *kite-level* (big picture)

4.27 Example

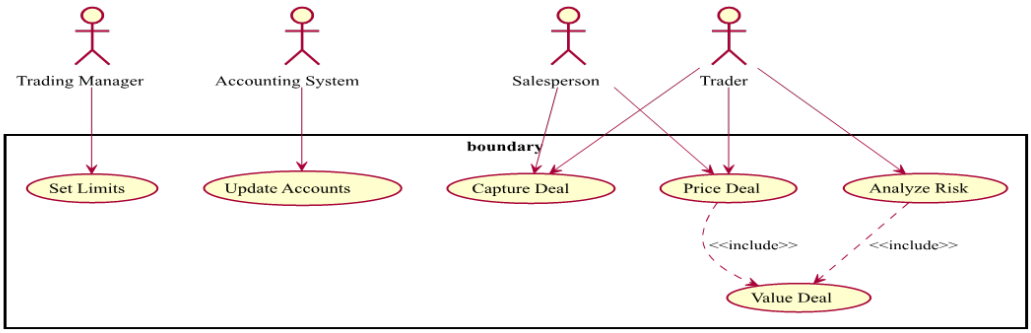


Figure 4.12: Use case diagram sample.

4.28 Remarks

- A use case is a set of scenarios tied together by a common user goal
- Actors do not need to be human
- The specification is surprisingly sparse on use cases
- The value lies completely in the content, not the diagram
- Usually one starts by writing a use case text
- Great way for brainstorming alternatives

4.29 Sequence diagrams

- Most important interaction diagram type
- Captures the behavior of a single scenario
- Shows example objects and the messages that are passed between these within the scenario (displays no control flow)
- Objects () are bound to lifelines (dashed lines)
- Messages have a direction
- The destruction (delete) of an object is shown with an X
- Additional annotations like *«new»* are possible

4.30 Example

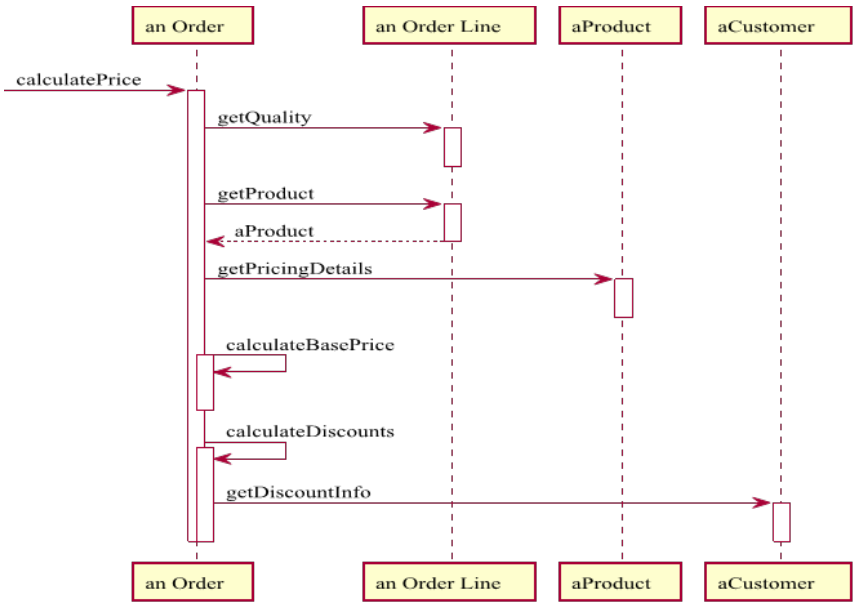


Figure 4.13: Sequence sample.

4.31 Remarks

- Common issue: How to show looping? Answer: You don't!
- If logic is required use an *interaction frame*, but only in extreme cases
- Sequence diagrams should illustrate how objects interact
- Even though deletion is not required in GC environments using the X to indicate disposable objects is worth it
- Asynchronous messages can also be displayed using the special arrow

4.32 Creation and Deletion

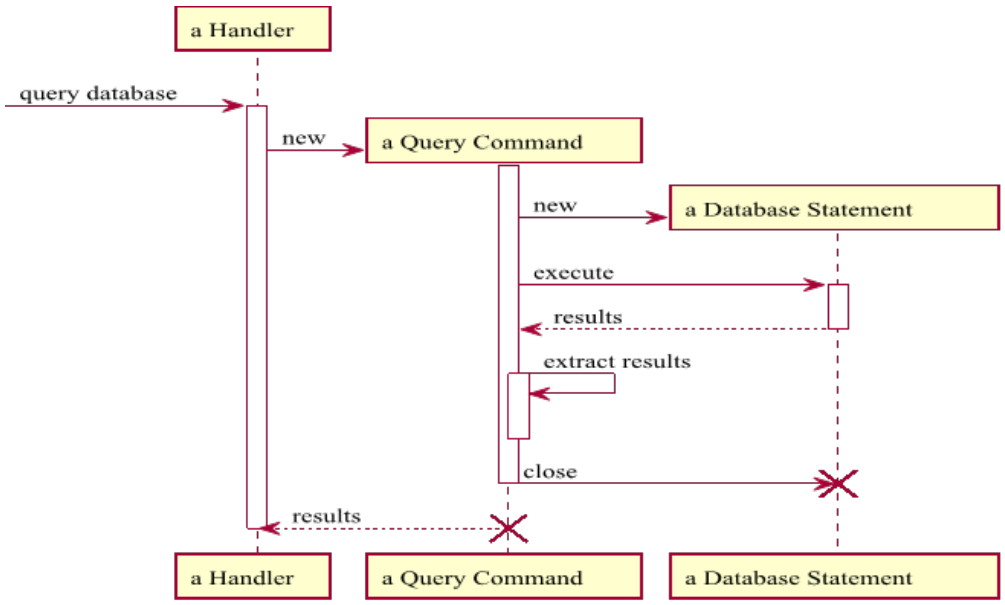


Figure 4.14: Sequence with creation and deletion.

4.33 Wait...

- Package diagrams can be used for illustrating top level library views
- Deployment diagrams are useful for setup or installation processes
- State machine diagrams supply everything for showing relations between (even concurrent) states
- Timing diagrams notate timing constraints which is important e.g. for electronic engineering
- Also communication and (composite) structure diagram types are existing
- Examples of some types to follow

4.34 State Machine diagram

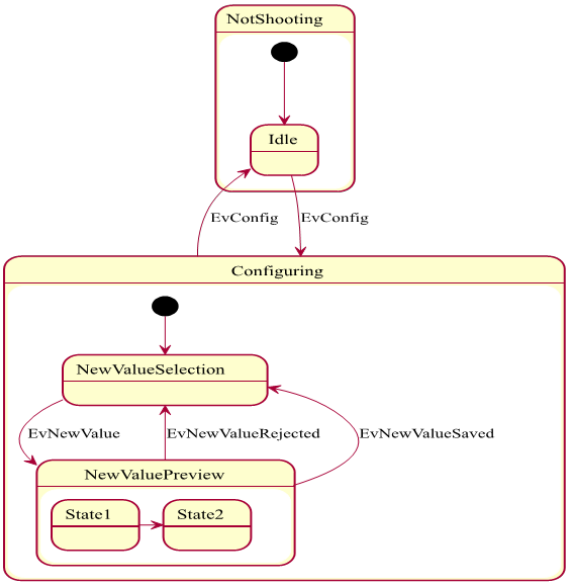


Figure 4.15: State sample.

4.35 Component diagram

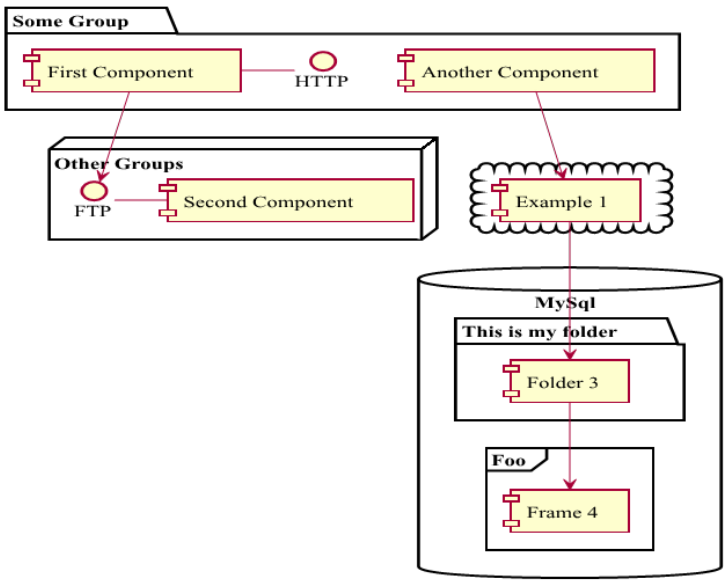


Figure 4.16: Component sample.

4.36 Nested classes sample

- Use a composition arrow (as defined for packages) to indicate nested classes, if really needed

- Here Enumerator is nested within Dictionary

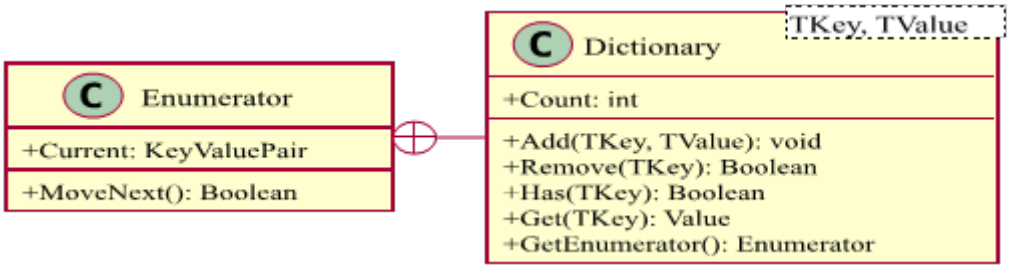


Figure 4.17: Nested classes.

4.37 Common conventions

- Command-Query separation: Operations with no return value commands, all others are queries (they do not modify anything)
- If relationships to specific class of objects will change we can mark the relationship in the class diagram with a «temporal» keyword
- Messages between classes can be drawn using arrows above the relationships with the name of the messages between these classes
- We can use a rake symbol within an action box to indicate a sub activity diagram in an activity diagram

4.38 References

- Michael L. Collard, SEM Lecture notes (`http://www.cs.uakron.edu/~collard/cs680/notes/`)
- yUML UML editor (`http://www.yuml.me`)
- PlantUML UML editor (`http://plantuml.sourceforge.net`)
- Wikipedia article (`http://en.wikipedia.org/wiki/Unified_Modeling_Language`)
- Example UML diagrams (`http://www.uml-diagrams.org/`)
- Introduction to UML 2.0 (`http://www.agilemodeling.com/essays/umlDiagrams.htm`)
- Tutorialspoint on class diagrams (`http://www.tutorialspoint.com/uml/uml_class_diagram.htm`)
- YouTube video on class diagrams (`http://www.youtube.com/watch?v=3cmzqZzwNDM`)

4.39 Literature

- Ambler, Scott William (2004). *The Object Primer: Agile Model Driven Development with UML 2.*
- Chonoles, Michael Jesse; James A. Schardt (2003). *UML 2 for Dummies.*
- Fowler, Martin (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.).*
- Jacobson, Ivar; Booch, Grady; Rumbaugh, James (1998). *The Unified Software Development Process.*

Chapter 5

Creational patterns

5.1 Introduction

- Deal with object creation
- Try to control the object creation process
- Basically two intentions are often seen here:
  1. Making systems independent of specific classes
  2. Hiding how classes are created and combined
- Goal: Minimizing dependencies and complexities while maximizing flexibility

5.2 The singleton pattern

- Restricts the instantiation of a class to one object
- Useful if only one instance is needed and this instance is used by a lot of objects
- Much better controlled than global variables
- Encapsulates the given set of variables
- Can be resource friendly by allocating memory when required, not before
- Required: Private or protected constructor and *static* readonly property to access instance

5.3 Singleton diagram

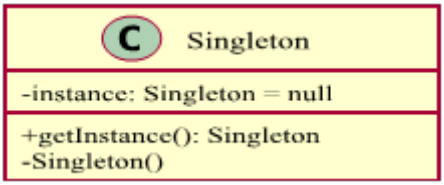


Figure 5.1: Singleton pattern.

5.4 Remarks

- There are multiple variants - the shown one is called **lazy initialization** and represents the most common approach
- By using a *static constructor* we could have an **eager initialization** and avoid the unnecessary condition in the property
- Therefore there is a way that uses both approaches to have the advantages of lazy loading without any repeating condition testing
- Another advantage of this solution is that it would be thread-safe (the original lazy is *not* thread-safe)

5.5 Lazy loading singleton

```
public class Singleton
{
   private static Singleton instance;

   private Singleton() {}

   public static Singleton Instance
   {
      get
      {
         if (instance == null)
            instance = new Singleton();

         return instance;
      }
   }
}
```

5.6 Practical considerations

- Quite often used together with other patterns
- Central location for variables (like a namespace), but more clean and encapsulated

- Can be controlled quite easily (getInstance could have more logic)
- Configuration of application stored in a Singleton allows easy and automated serialization and deserialization of stored values
- APIs make great use of singletons to provide a single entry point
- Enhance a resource friendly way of using particular objects

5.7 Global application state

```
public sealed class ApplicationState
{
   private static ApplicationState _instance;
   private static object _lockThis = new object();

   private ApplicationState() { }

   public static ApplicationState Instance
   {
      get
      {
         if (_instance == null)
         {
            lock (_lockThis)
            {
               if (_instance == null)
                  _instance = new ApplicationState();
            }
         }
         return _instance;
      }
   }

   public string LoginId { get; set; }

   public int MaxSize { get; set; }
}
```

5.8 The prototype pattern

- Use exchangeable objects as parts of a class, instead of deriving from another class
- Big advantage: We do not have to destroy the complete object to change the *parent* (which is now the *prototype*)
- Some languages are actually using this pattern implicitly
- The creation of the prototype is usually done by calling a cloning method on an already existing instance
- The cloning process should result in a **deep** copy of the prototype

5.9 Prototype diagram

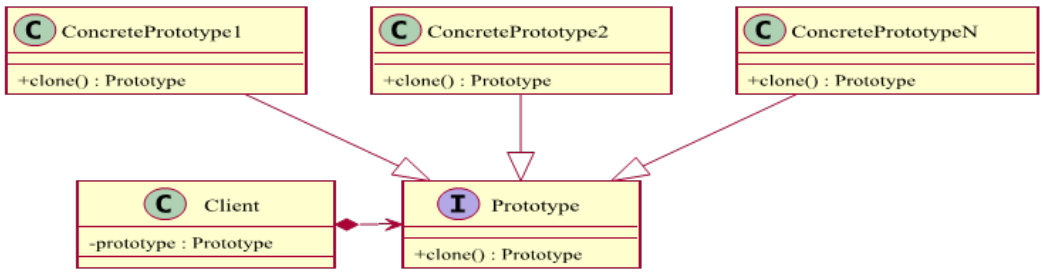


Figure 5.2: Prototype pattern.

5.10 Remarks

- Cloning is an important part, but it should not be the only reason to follow the prototype pattern
- .NET and Java have already a single purpose interface for cloning alone
- The main reason for this pattern is flexibility
- Prototype doesn't require subclassing, but an *initialize* operation
- One instance of a class is used as a breeder of all future instances
- Prototype is unique among the other creational patterns since it only requires an object, not a class

5.11 Base class prototype

```
abstract class Prototype
{
    private readonly string _id;

    protected Prototype(string id)
    {
        _id = id;
    }

    public string Id
    {
        get { return _id; }
    }

    public abstract Prototype Clone();
}

class ConcretePrototype : Prototype
{
    public ConcretePrototype(string id)
        : base(id)
```

```
    {
    }

    public override Prototype Clone()
    {
        return (Prototype)MemberwiseClone();
    }
}
```

5.12 Practical considerations

- The prototype pattern can save resources by cloning only certain parts and omitting some initialization
- Usually we will get a general object back, so casting might be required
- Designs that make heavy use of the other patterns (e.g. composite, decorator, ...) often can benefit from prototype as well
- A Singleton that contains all prototype objects is quite useful
- The singleton would then have a method to create (clone) a specific prototype

5.13 Changing a template

```
public abstract class Template
{
    public abstract Template Copy();

    public Color Foreground { get; set; }

    public Color Background { get; set; }

    public string FontFamily { get; set; }

    public double FontSize { get; set; }

    /* ... */

    protected static void CopyBasicFields(Template original, Template copy)
    {
        copy.Foreground = original.Foreground;
        /* ... */
    }
}

public class DefaultTemplate : Template
{
    public DefaultTemplate()
    {
        Foreground = Color.FromRgb(0, 0, 0);
        Background = Color.FromRgb(255, 255, 255);
        FontFamily = "Arial";
        FontSize = 12.0;
```

```
    }

    public override Template Copy()
    {
        DefaultTemplate copy = new DefaultTemplate();
        CopyBasicFields(this, copy);
        /* ... */
        return copy;
    }
}

public class Document
{
    Template _template;

    public Document(Template template)
    {
        ChangeTemplate(template);
    }

    public void ChangeTemplate(Template template)
    {
        _template = template.Copy();
    }
}
```

5.14 The factory method pattern

- Create specific classes without using them
- Idea: Tell a central factory which class to create (e.g. by using a string)
- The factory then knows which class to use and how to create
- Goal: Maintainability and extendability gain
- Using *DRY*: Creation code on a single central place
- *Information hiding* is possible if special information is required that only the factory needs to know about

5.15 Factory method diagram

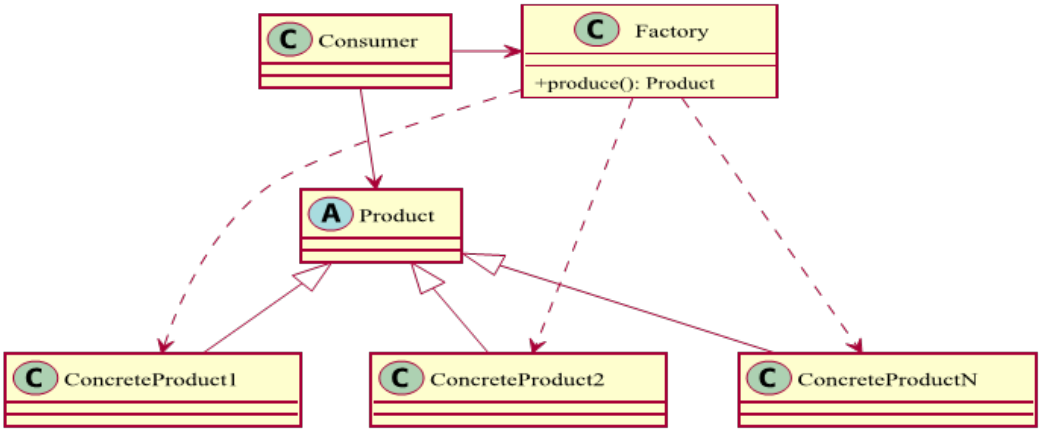


Figure 5.3: Factory method pattern.

5.16 Remarks

- The factory method exists in a lot of variations and makes most sense with otherwise complicated APIs or library extensions
- Generally one might think of two more methods in the factory to register and unregister new products with instructions on how to create
- Languages like C# or Java have *Reflection* built-in, which can enable automatically constructed factories
- Quite often the factory method pattern is displayed with an abstract class for the factory, but actually this is not required

5.17 String based factory

```
public abstract class SmartPhone
{
    public abstract double Price { get; }

    /* ... */
}
class GalaxyS3 : SmartPhone
{
    public override double Price
    {
        get { return 599.99; }
    }

    /* ... */
}
public class Factory
{
```

```
    public SmartPhone Produce(String type)
    {
        switch (type)
        {
            case "GalaxyS3":
                return new GalaxyS3();
            /* ... */
            default:
                return null;
        }
    }
}
class Consumer
{
    private SmartPhone phone;
    private double money;

    public void BuyNewSmartPhone(Factory factory, String type)
    {
         phone = factory.Produce(type);
         money -= phone.Price;
    }
}
```

5.18 Practical considerations

- Using the singleton pattern for the factory is often useful
- If the products follow the prototype pattern then object creation (and registration, if provided) is really simple and straight forward
- One of the most famous factories can be found in the browser: the DOM only provides access for creating elements over a factory
- Actually we use already a factory by typing in addresses (scheme)
- Games quite often make use of factories to produce items
- The provided abstraction of factories help to reduce dependencies

5.19 A factory for commands

```
interface ICommand
{
    string Info { get; }

    string Help { get; }

    bool CanUndo { get; }

    string[] Calls { get; }

    string FlushOutput();

    bool Invoke(string command, string[] arguments);
```

```
    void Undo();
}

class Commands
{
    static Commands instance;

    ICommand[] commands;
    Stack<ICommand> undoList;

    private Commands()
    {
        undoList = new Stack<ICommand>();
        undoCallings = new List<string>();
        commands = Assembly.GetExecutingAssembly()
            .GetTypes()
            .Where(m => !m.IsInterface && typeof(ICommand).IsAssignableFrom(m))
            .Select(m => m.GetConstructor(System.Type.EmptyTypes).Invoke(null)
                as ICommand)
            .ToArray();
    }

    public static Commands Instance
    {
        get { return instance ?? (instance = new Commands()); }
    }

    public ICommand Create(string command, params string[] args)
    {
        foreach (ICommand cmd in commands)
        {
            if (CheckForCall(cmd, command))
                return cmd;
        }

        return null;
    }

    public bool Invoke(string command, params string[] args)
    {
        ICommand cmd = Create(command, args);

        if (cmd != null && cmd.Invoke(command, args))
        {
            if (cmd.CanUndo)
                undoList.Push(cmd);

            return true;
        }

        return false;
    }
}
```

5.20 Discussion

- Singletons are sometimes overused, but can be used in many situations

- The factory method is particularly useful to decouple dependencies on children of classes
- The ability to clone itself is interesting especially if construction would be impossible otherwise (due to dependencies or language constraints)
- Decoupling the factory can also be very beneficial, which results in the next pattern: The *abstract factory pattern*

5.21 The abstract factory pattern

- What if we need specializations of outcoming products?
- Goal: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Instead of wanting *Product* instances we want *ProductVariant* objects, with specialized implementations
- However, we do not need to know which factory is producing it
- This decouples a concrete factory from our code and enables application aware productions

5.22 Abstract factory diagram

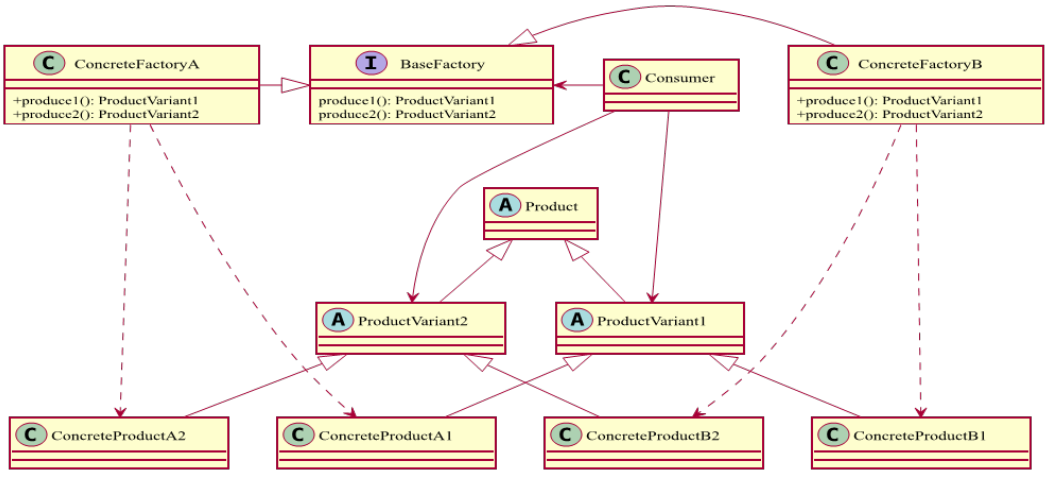


Figure 5.4: Abstract factory pattern.

5.23 Remarks

- Abstract factories allow cross-platform development
- Trick: Extract concrete factory and products into adapter library
- Exchange adapter library depending on platform

- There could also be a factory of factories, using both, the Factory method and the abstract factory pattern
- Abstract factories in this form are not as popular as a plain factory
- Most popular is a hybrid approach, that takes an abstract factory (e.g. defined in an interface), to produce a single type of product

5.24 Method based abstract factory

```
public abstract class Control
{
    protected abstract void PaintControl();
}
public abstract class Button : Control
{
}
public abstract class TextBox : Control
{
}
class WinButton : Button
{
    protected override void PaintControl()
    {
        /* Implementation */
    }
}
class WinTextBox : TextBox
{
    protected override void PaintControl()
    {
        /* Implementation */
    }
}
class OsxButton : Button
{
    protected override void PaintControl()
    {
        /* Implementation */
    }
}
class OsxTextBox : TextBox
{
    protected override void PaintControl()
    {
        /* Implementation */
    }
}
public interface IFactory
{
    Button CreateButton();
    TextBox CreateTextBox();
}
public class WinFactory : IFactory
{
    public Button CreateButton()
```

```
    {
        return new WinButton();
    }

    public TextBox CreateTextBox()
    {
        return new WinTextBox();
    }
}
public class OsxFactory : IFactory
{
    public Button CreateButton()
    {
        return new OsxButton();
    }

    public TextBox CreateTextBox()
    {
        return new OsxTextBox();
    }
}
```

5.25 Practical considerations

- The abstract factory pattern is quite often used with dependency injection (DI), which will be introduced later
- Here the concrete products will be resolved automatically
- Also the factory can then be resolved automatically
- This yields a maximum an flexibility and minimizes the coupling
- There are complete frameworks like ADO.NET build upon this
- Idea: Abstract the interface to the database and hide concrete implementations for specific databases (like MySQL, MariaDB, ...)

5.26 ADO.NET abstract factory

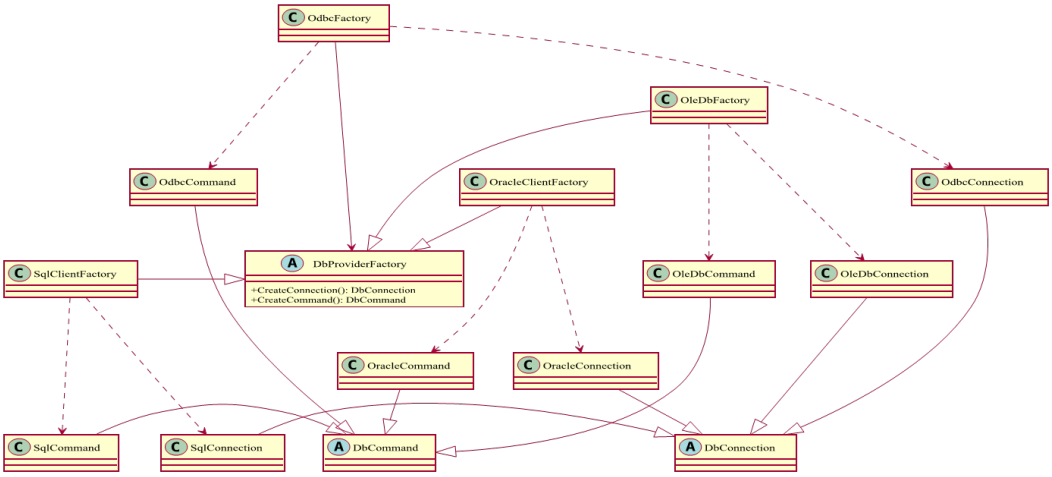


Figure 5.5: The abstract factory pattern in ADO.NET.

5.27 The builder pattern

- Quite often a defined sequence of method calls is required for correctly initializing objects
- This scenario goes far beyond what a constructor should deliver
- Idea: Create a pattern, which follows the right sequence by design
- The **builder pattern** enables the step-by-step creation of complex objects using the correct sequence of actions
- Here the construction is controlled by a *director object* that only needs to know the type of object it is to create

5.28 Builder diagram

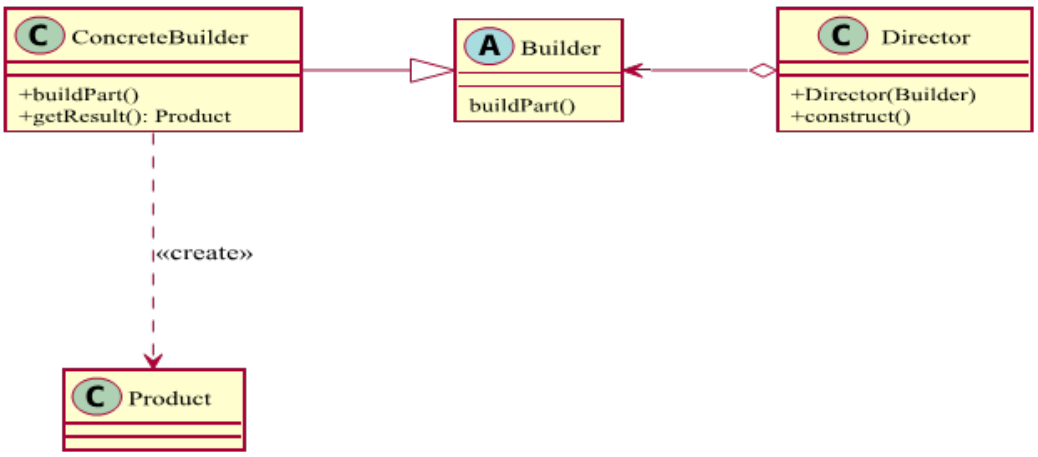


Figure 5.6: Builder pattern.

5.29 Remarks

- The flow requires that a calling object needs to know the concrete builder and the director
- In general the directory does not require to get the builder over the constructor, it could also be set by using a property
- However, the director requires a builder, instance which is reflected by such a design
- The director handles the whole construction process
- In the end one just has to call getResult() on the chosen builder

5.30 Fluent interface

- The builder pattern is a solution to the telescoping constructor problem
- Instead of using numerous constructors, the builder pattern uses another object, that receives each initialization parameter step by step
- Builders are good candidates for a fluent API using the fluent interface
- This can build upon **chaining**, i.e. we always return something *useful* (if nothing would be returned, then the current instance should be returned)

5.31 Simple builder

```
public interface IBuilder
{
    void BuildPartA();
    void BuildPartB();
    Product Result() { get; }
}
static class Director
{
    public static void Construct(IBuilder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}
class ConcreteBuilder : IBuilder
{
    private Product _product;

    public ConcreteBuilder
    {
        _product = new Product();
    }

    public void BuildPartA()
    {
        /* ... */
    }

    public void BuildPartB()
    {
        /* ... */
    }

    public Product Result
    {
        get { return _product; }
    }
}
class Product
{
    /* ... */
}
```

5.32 Practical considerations

- Quite often a *static* class is enough, with the builder dependency being shifted to the construction function
- The director might also create intermediate objects, which can then be implicitly buffered or destroyed
- Sometimes there is no specific director class, but a director method
- This could be done to centralize the director and minimize dependencies
- An example would be the construction of database commands

5.33 Database command builder

```
public abstract class DbCommandBuilder
{
    public abstract DbCommand GetDeleteCommand();
    /* ... */
}
public class SqlCommandBuilder : DbCommandBuilder
{
   private DbCommand BuildDeleteCommand(DataTableMapping mappings, DataRow
       dataRow)
    {
        DbCommand command = this.InitializeCommand(this.DeleteCommand);
        StringBuilder builder = new StringBuilder();
        int parameterCount = 0;
        builder.Append("DELETE FROM ");
        builder.Append(this.QuotedBaseTableName);
        parameterCount = this.BuildWhereClause(
        mappings, dataRow, builder, command, parameterCount, false);
        command.CommandText = builder.ToString();
        RemoveExtraParameters(command, parameterCount);
        this.DeleteCommand = command;
        return command;
    }

    public SqlCommandBuilder(SqlDataAdapter da)
    {
        /* set some properties */
    }

    public override DbCommand GetDeleteCommand()
    {
        DataTableMapping mappings = ComputeMappings();
        DataRow row = ComputeRow();
        return BuildDeleteCommand(mappings, row);
    }
}
```

5.34 References

- CodeProject: Design Patterns 1 of 3 (http://www.codeproject.com/Articles/430590/Design-Patterns-1-of-3-Creational-Design-Patterns)
- Wikipedia: Prototype pattern (http://en.wikipedia.org/wiki/Prototype_pattern)
- Wikipedia: Creational pattern (http://en.wikipedia.org/wiki/Creational_pattern)
- DoFactory: Builder pattern (http://www.dofactory.com/Patterns/PatternBuilder.aspx)
- Wikipedia: Fluent interface (http://en.wikipedia.org/wiki/Fluent_interface)
- How I explained Design Patterns to my wife: Part 1 (http://www.codeproject.com/Articles/98598/How-I-explained-Design-Patterns-to-my-wife-Part-1)

5.35 Literature

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object Oriented Software.*
- Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). *Head First Design Patterns.*
- Fowler, Martin; Beck, Kent; Brant, John; Opdyke, William; Roberts (1999). *Refactoring: Improving the Design of Existing Code.*
- McConnell, Steve (2004). "Design in Construction". *Code Complete.*

Chapter 6

Behavioral patterns

6.1 Introduction

- Deal with communication between objects
- Try to identify common communication
- Intentions for implementing such patterns:
  1. Increased flexibility
  2. Determined program flow
  3. Optimize resource usage
- Sometimes patterns introduced here are already included in languages

6.2 The observer pattern

- How to notify an unknown amount of unknown objects with a specific message?
- The solution to this question is the observer pattern
- An object, called *subject*, is maintaining a list of dependents
- They are called *observers* and contain a known method
- This method is invoked once the state changes
- The invocation is usually called *firing*, with the message being an *event*

6.3 Observer diagram

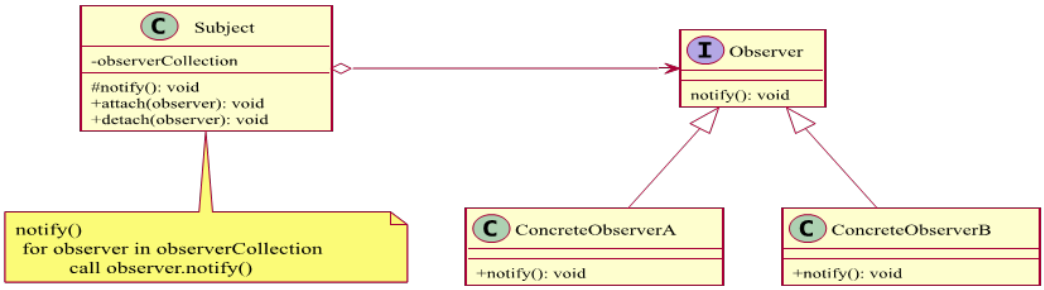


Figure 6.1: Observer pattern.

6.4 Remarks

- Key part of MVC (Model View Controller) architecture (introduced later)
- First implemented in Smalltalk, which introduced MVC and OOP in general and was used to create the first OS with a GUI
- Almost all GUI frameworks contain this pattern
- C# has it inbuilt using the event keyword
- In Java java.util.Observable contains Observable
- This pattern enables de-coupled messaging

6.5 Anonymous mute observer

```
public class Subject
{
    private List<IObserver> _observers;
    private string _state;

    public Subject()
    {
        _observers = new List<IObserver>();
    }

    public string State
    {
        get { return _state; }
        set
        {
            _state = value;
            Notify();
        }
    }

    public void Attach(IObserver o)
    {
```

```
        _observers.Add(o);
    }

    public void Detach(IObserver o)
    {
        _observers.Remove(o);
    }

    protected void Notify()
    {
        foreach (IObserver o in _observers)
        {
            o.Update(this);
        }
    }
}
public interface IObserver
{
    void Update(object sender);
}
public class ConcreteObserver : IObserver
{
    public void Update(object sender)
    {
        /* Do something with the sender */
    }
}
```

6.6 Practical considerations

- Practically the observer pattern has some implementation dependent flaws (e.g. a framework or a custom implementation)
- Main problem: How to cleanly detach an event listener?
- In managed languages memory leaks can occur
- Here weak references provide a way out of this mess
- In native languages segmentation faults are possible
- Hence: Always think about how (and when) to remove the observer

6.7 A stock ticker

```
public struct Stock
{
    public double Price { get; set; }
    public string Code { get; set; }
}
public interface IStockObserverBase
{
    void Notify(Stock stock);
}
public class SpecificStockObserver : IStockObserverBase
```

```
{
    public SpecificStockObserver(string name)
    {
        Name = name;
    }

    public string Name
    {
        get;
        private set;
    }

    public void Notify(Stock stock)
    {
        if(stock.Code == Name)
            Console.WriteLine("{0} changed to {1:C}", stock.Code, stock.Price);
    }
}
public abstract class StockTickerBase
{
    readonly protected List<IStockObserverBase> _observers = new List<
        IStockObserverBase>();

    public void Register(IStockObserverBase observer)
    {
        if(!_observers.Contains(observer))
        {
            _observers.Add(observer);
        }
    }

    public void Unregister(IStockObserverBase observer)
    {
        if (_observers.Contains(observer))
        {
            _observers.Remove(observer);
        }
    }
}
public class StockTicker : StockTickerBase
{
    private List<Stock> stocks = new List<Stock>();

    public void Change(string code, double price)
    {
        Stock stock = new Stock
        {
            Code = code,
            Price = price
        };

        if (stocks.Contains(stock))
            stocks.Remove(stock);

        stocks.Add(stock);
        Notify(stock);
    }
```

```
    void Notify(Stock stock)
    {
        foreach (var observer in _observers)
        {
            observer.Notify(stock);
        }
    }
}
```

6.8 The command pattern

- Commands are important and exist in various forms
- They could be command line arguments or input, events, ...
- Today they are mostly present by clicking on buttons in GUI
- The command pattern defines four terms:
  - *command* (the central object)
  - *receiver* (contains specific method(s))
  - *invoker* (uses the command for invocation)
  - *client* (knows when to pass the command to the invoker)

6.9 An illustrative example

- As an example let's take a simple application with a GUI
- The client would be a Button
- There are multiple invokers like a Click
- Each invoker has a number of commands defined
- A command connects the defined invoker of an existing Button with our own implementation (receiver)
- The command is therefore something like an abstraction of the underlying implementation

6.10 Command diagram

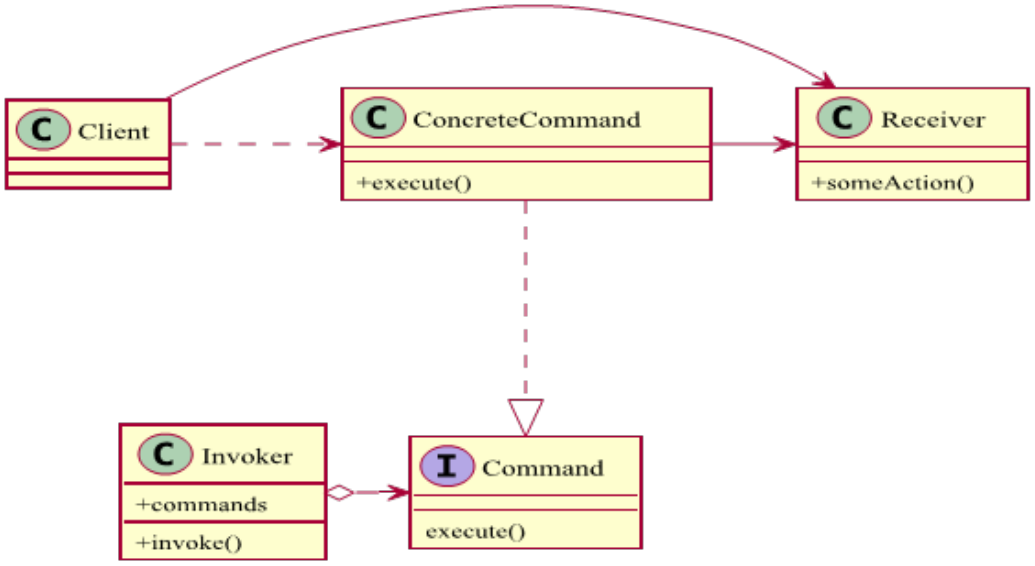


Figure 6.2: Command pattern.

6.11 Remarks

- The command pattern is quite similar to the factory method, but instead of creation it is about execution
- The terminology is not consistent and often confusing
- Implementations might consider having *do* and *undo* instead of *execute*
- Also a Boolean indicator if the command can be executed might make sense (usually this is a get method)
- Commands are a great extension to the observer pattern

6.12 Light commands

```
public interface ICommand
{
    void Execute();
}
public class Switch
{
    private List<ICommand> _commands = new List<ICommand>();

    public void StoreAndExecute(ICommand command)
    {
        _commands.Add(command);
        command.Execute();
    }
}
```

```
public class Light
{
    public void TurnOn()
    {
        Console.WriteLine("The light is on");
    }

    public void TurnOff()
    {
        Console.WriteLine("The light is off");
    }
}
public class FlipUpCommand : ICommand
{
    private Light _light;

    public FlipUpCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.TurnOn();
    }
}
public class FlipDownCommand : ICommand
{
    private Light _light;

    public FlipDownCommand(Light light)
    {
        _light = light;
    }

    public void Execute()
    {
        _light.TurnOff();
    }
}
```

6.13 Practical considerations

- There are some points when the command pattern is really useful:
  - A history of requests is needed
  - Callback functionality is required
  - Requests are handled at variant times / orders
  - Invoker and client should be decoupled
- Commands are useful for wizards, progress bars, GUI buttons, menu actions, and other transactional behavior

6.14 Controlling a robot

```
public abstract class RobotCommandBase
{
    protected Robot _robot;

    public RobotCommandBase(Robot robot)
    {
        _robot = robot;
    }

    public abstract void Execute();

    public abstract void Undo();
}
public class MoveCommand:RobotCommandBase
{
    public int ForwardDistance { get; set; }

    public MoveCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.Move(ForwardDistance);
    }

    public override void Undo()
    {
        _robot.Move(-ForwardDistance);
    }
}
public class RotateLeftCommand : RobotCommandBase
{
    public double LeftRotationAngle { get; set; }

    public RotateLeftCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.RotateLeft(LeftRotationAngle);
    }

    public override void Undo()
    {
        _robot.RotateRight(LeftRotationAngle);
    }
}
public class RotateRightCommand : RobotCommandBase
{
    public double RightRotationAngle { get; set; }

    public RotateRightCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.RotateRight(RightRotationAngle);
    }

    public override void Undo()
    {
```

```
        _robot.RotateLeft(RightRotationAngle);
    }
}
public class TakeSampleCommand : RobotCommandBase
{
    public bool TakeSample { get; set; }

    public TakeSampleCommand(Robot robot) : base(robot) { }

    public override void Execute()
    {
        _robot.TakeSample(true);
    }

    public override void Undo()
    {
        _robot.TakeSample(false);
    }
}
public class RobotController
{
    private Queue<RobotCommandBase> commands;
    private Stack<RobotCommandBase> undoStack;

    public RobotController()
    {
        commands = new Queue<RobotCommandBase>();
        undoStack = new Stack<RobotCommandBase>();
    }

    public void EnqueueCommand(RobotCommandBase command)
    {
        commands.Enqueue(command);
    }

    public void ClearCommands()
    {
        commands.Clear();
    }

    public void ExecuteAllCommands()
    {
        Console.WriteLine("EXECUTING COMMANDS.");

        while (commands.Count > 0)
        {
            RobotCommandBase command = commands.Dequeue();
            command.Execute();
            undoStack.Push(command);
        }
    }

    public void UndoCommands(int numUndos)
    {
        Console.WriteLine("REVERSING {0} COMMAND(S).", numUndos);

        while (numUndos > 0 && undoStack.Count > 0)
        {
```

```
            RobotCommandBase command = undoStack.Pop();
            command.Undo();
            numUndos--;
        }
    }
}
public class Robot
{
    public void Move(int distance)
    {
        if (distance > 0)
            Console.WriteLine("Robot moved forwards {0}mm.", distance);
        else
            Console.WriteLine("Robot moved backwards {0}mm.", -distance);
    }

    public void RotateLeft(double angle)
    {
        if (angle > 0)
            Console.WriteLine("Robot rotated left {0} degrees.", angle);
        else
            Console.WriteLine("Robot rotated right {0} degrees.", -angle);
    }

    public void RotateRight(double angle)
    {
        if (angle > 0)
            Console.WriteLine("Robot rotated right {0} degrees.", angle);
        else
            Console.WriteLine("Robot rotated left {0} degrees.", -angle);
    }

    public void TakeSample(bool take)
    {
        if(take)
            Console.WriteLine("Robot took sample");
        else
            Console.WriteLine("Robot released sample");
    }
}
```

6.15 The chain-of-responsibility pattern

- Quite often we want to execute various commands in a certain way
- Creating a chain of responsibility ensures a loosely coupled system
- The chain uses a source of (atomic) commands a series of processing objects, which contain the logic
- Classically the pattern defines a linked list of handlers
- The direction is not fixed, it could also be a *tree of responsibility*
- In general this is a perfect extension to the command pattern

6.16 Chain-of-responsibility diagram

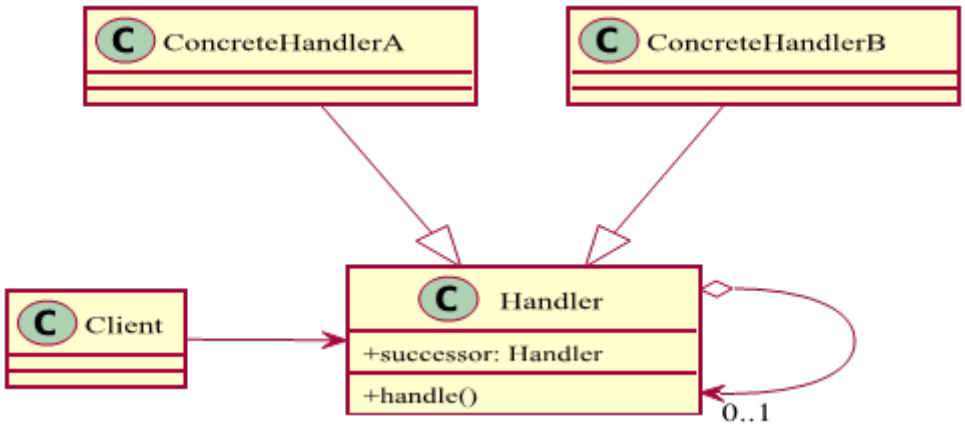


Figure 6.3: Chain-of-responsibility pattern.

6.17 Remarks

- The command is passed to the first processing object which can handle this command or send to its successor
- A single handler only needs to know its successor, if any
- This is a big plus, but might lead to a circle (and infinite loop)
- Also the chain is only as good as its weakest member
- This means that if the last handler is not responsible for the request, it will not execute the build chain of commands

6.18 A simple sample

```
public abstract class HandlerBase
{
    public HandlerBase Successor
    {
        get;
        set;
    }

    public abstract void HandleRequest(int request);
}
public class ConcreteHandlerA : HandlerBase
{
    public override void HandleRequest(int request)
    {
        if (request == 1)
            Console.WriteLine("Handled by ConcreteHandlerA");
        else if (Successor != null)
            Successor.HandleRequest(request);
```

```
    }
}
public class ConcreteHandlerB : HandlerBase
{
    public override void HandleRequest(int request)
    {
        if (request > 10)
            Console.WriteLine("Handled by ConcreteHandlerB");
        else if (Successor != null)
            Successor.HandleRequest(request);
    }
}
```

6.19 Practical considerations

- Use this pattern if more than one handler for a request is available
- Otherwise if one handler might require another handler
- Or if the set of handlers varies dynamically
- Chain-of-responsibility patterns are great for filtering
- The biggest advantage is the extensibility
- Also the specific handler does not have to be known (information hiding)

6.20 Chain-of-responsibility Vs Command

	Chain-of-responsibility	Command
Client creates	Handlers	Commands
Variations of	Handlers	Commands, receivers
Clients can use	Multiple handlers	Different receivers
Client calls	Handlers	Receivers
Work done in	Handler	Receiver
Decisions based on	Limits in handlers	Routing in commands
Unknown requests?	Received, not handled	Executes nothing

6.21 A coin machine

```
public class Coin
{
    public float Weight { get; set; }
    public float Diameter { get; set; }
}
public enum CoinEvaluationResult
{
    Accepted,
    Rejected
}
public abstract class CoinHandlerBase
{
    protected CoinHandlerBase _successor;
```

```
    public void SetSuccessor(CoinHandlerBase successor)
    {
        _successor = successor;
    }

    public abstract CoinEvaluationResult EvaluateCoin(Coin coin);
}
class FiftyPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 8) < 0.02 && Math.Abs(coin.Diameter - 27.3) <
             0.15)
        {
            Console.WriteLine("Captured 50p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}
class FivePenceHandler : CoinHandlerBase
{

    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 3.25) < 0.02 && Math.Abs(coin.Diameter - 18)
            < 0.1)
        {
            Console.WriteLine("Captured 5p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}
class OnePoundHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 9.5) < 0.02 && Math.Abs(coin.Diameter - 22.5)
             < 0.13)
        {
            Console.WriteLine("Captured GBP1");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
```

```
    }
}
class TenPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 6.5) < 0.03 && Math.Abs(coin.Diameter - 24.5)
             < 0.15)
        {
            Console.WriteLine("Captured 10p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}
class TwentyPenceHandler : CoinHandlerBase
{
    public override CoinEvaluationResult EvaluateCoin(Coin coin)
    {
        if (Math.Abs(coin.Weight - 5) < 0.01 && Math.Abs(coin.Diameter - 21.4) <
             0.1)
        {
            Console.WriteLine("Captured 20p");
            return CoinEvaluationResult.Accepted;
        }
        if (_successor != null)
        {
            return _successor.EvaluateCoin(coin);
        }
        return CoinEvaluationResult.Rejected;
    }
}
```

6.22 The iterator pattern

- Sometimes we want to iterate over a bunch of objects
- This is actually like handling a linked list, e.g. the previous pattern
- Traversing lists, trees, and other structures is important
- Key: Iterating without knowing the explicit structure
- Goal: Provide a simple interface for traversing a collection of items
- Languages like C# or Java have the iterator pattern build in

6.23 Iterator diagram

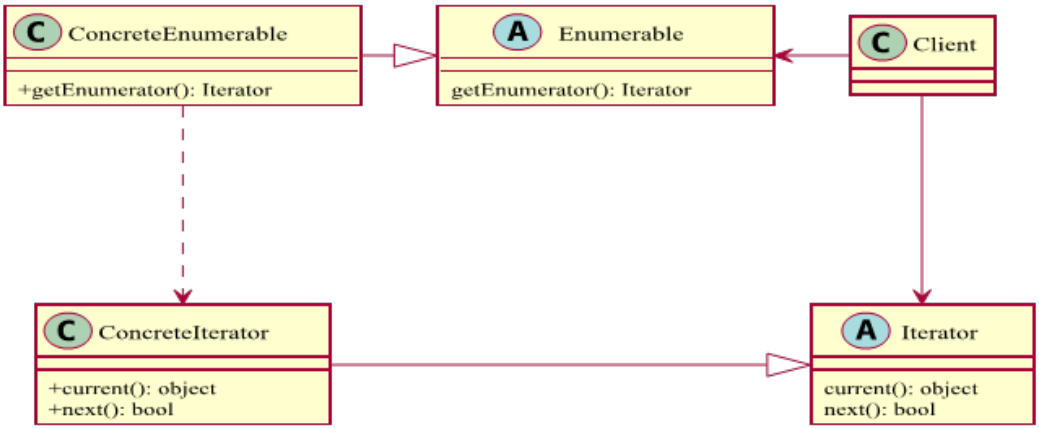


Figure 6.4: Iterator pattern.

6.24 Remarks

- In C++ the pattern is usually implemented with pointer operators
- Java and C# already provide an interface to implement
- Usually there is a generic interface, which should be used
- The iterator pattern enables technologies such as LINQ
- Passing iterators as arguments can be very efficiently
- Reason: The iterator object maintains the state of the iteration

6.25 Trivial iterator

```
interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

interface IEnumerable
{
    IEnumerator GetEnumerator();
}

class MyClass : IEnumerable
{
    object[] array;

    /* ... */
```

```
    public IEnumerator GetEnumerator()
    {
        return new ArrayIterator(array);
    }
}

class ArrayIterator : IEnumerator
{
    int index;
    object[] array;

    public ArrayIterator (object[] array)
    {
    this.array = array;
    Reset();
    }

    public object Current
    {
        get { return index >= 0 && index < array.Length ? array[index] : null; }
    }

    public bool MoveNext()
    {
        index++;
        return Current != null;
    }

    public void Reset()
    {
        index = -1;
    }
}
```

6.26 Practical considerations

- C# and Java have a loop construct based on the iterator pattern
- Here GetEnumerator() is called implicitely
- Then MoveNext() is used until false is returned
- This makes foreach() and for(:) more expensive than the classic for loop, which does not require any function calls
- Advantage: We can iterate over arbitrary objects
- In C++ we can always use pointer overloads to achieve this

6.27 Visitor pattern

- Separates a set of structured data from the functionality
- Promotes loose coupling
- Enables additional operations

- However, the data model (structure) is therefore really limited
- Basically it is just a container, which requires somebody to visit and evaluate its data

6.28 Visitor diagram

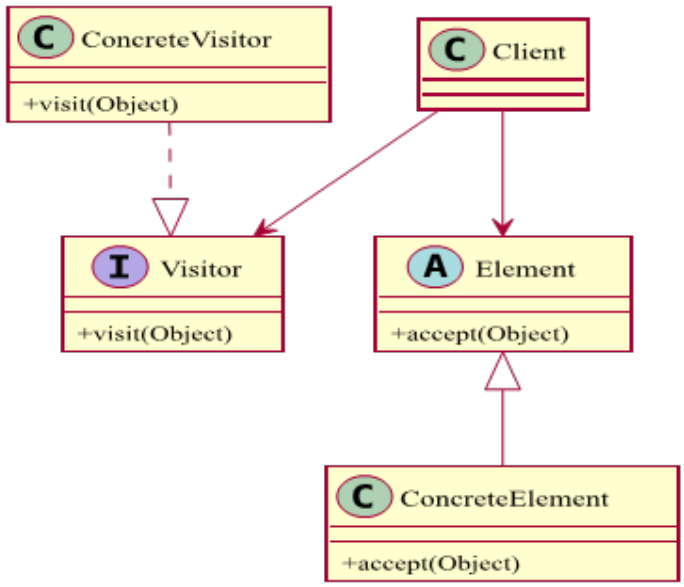


Figure 6.5: Visitor pattern.

6.29 Remarks

- New operations are added by creating new visitors
- Similar operations are managed in one visitor and separated from others
- A visitor can work over multiple class hierarchies
- However, the nice extensibility with visitors comes with greater constraints for the specific elements
- Compilers usually traverse the object tree via the visitor pattern
- In general the visitor pattern helps to apply operations on structures without information on the structure

6.30 Visitor sample

```
abstract class Visitor
{
    public abstract void VisitConcreteElementA(ConcreteElementA
        concreteElementA);
```

```
    public abstract void VisitConcreteElementB(ConcreteElementB
        concreteElementB);
}

class ConcreteVisitor : Visitor
{
    public override void VisitConcreteElementA(ConcreteElementA
        concreteElementA)
    {
        /* ... */
    }

    public override void VisitConcreteElementB(ConcreteElementB
        concreteElementB)
    {
        /* ... */
    }
}

abstract class Element
{
    public abstract void Accept(Visitor visitor);
}

class ConcreteElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }
}

class ConcreteElementB : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementB(this);
    }
}

class ObjectStructure
{
    private List<Element> _elements;

    public ObjectStructure()
    {
        _elements = new List<Element>();
    }

    public void Attach(Element element)
    {
        _elements.Add(element);
    }

    public void Detach(Element element)
    {
        _elements.Remove(element);
```

```
    }

    public void Accept(Visitor visitor)
    {
        foreach (Element element in _elements)
        {
            element.Accept(visitor);
        }
    }
}
```

6.31 Practical considerations

- The iterator pattern and visitor pattern has the same benefit, they are used to traverse object structures
- The visitor pattern can be used on complex structure such as hierarchical structures or composite structures
- Drawback: If a new visitable object is added to the framework structure all the implemented visitors need to be modified
- Part of the dependency problems can be solved by using reflection with a performance cost

6.32 Operations on employees

```
interface IVisitor
{
    void Visit(Element element);
}

class IncomeVisitor : IVisitor
{
    public void Visit(Element element)
    {
        Employee e = element as Employee;

        if (e != null)
        {
            e.Income *= 1.10;
            Console.WriteLine("{0} {1}'s new income: {2:C}", e.GetType().Name, e
                .Name, e.Income);
        }
    }
}

class VacationVisitor : IVisitor
{
    public void Visit(Element element)
    {
        Employee e = element as Employee;

        if (e != null)
        {
```

```
            Console.WriteLine("{0} {1}'s new vacation days: {2}", e.GetType().
                Name, e.Name, e.VacationDays);
        }
    }
}

abstract class Element
{
    public abstract void Accept(IVisitor visitor);
}

class Employee : Element
{
    private string _name;
    private double _income;
    private int _vacationDays;

    public Employee(string name, double income, int vacationDays)
    {
        this._name = name;
        this._income = income;
        this._vacationDays = vacationDays;
    }

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public double Income
    {
        get { return _income; }
        set { _income = value; }
    }

    public int VacationDays
    {
        get { return _vacationDays; }
        set { _vacationDays = value; }
    }

    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}

class Employees
{
    private List<Employee> _employees;

    public Employees()
    {
        _employees = new List<Employee>();
    }

    public void Attach(Employee employee)
```

```
    {
        _employees.Add(employee);
    }

    public void Detach(Employee employee)
    {
        _employees.Remove(employee);
    }

    public void Accept(IVisitor visitor)
    {
        foreach (Employee e in _employees)
        {
            e.Accept(visitor);
        }
    }
}

class Clerk : Employee
{
    public Clerk() : base("Hank", 25000.0, 14)
    {
    }
}

class Director : Employee
{
    public Director() : base("Elly", 35000.0, 16)
    {
    }
}

class President : Employee
{
    public President() : base("Dick", 45000.0, 21)
    {
    }
}
```

6.33 State pattern

- Classes have responsibilities and contain logic, but sometimes the logic can be messy
- A special case is a state-machine, where most logic is dependent on the current state
- The state pattern tries to simplify such a logic
- Idea: Encapsulate the state-dependent logic units as classes
- This allows to change behavior at run-time without changing the interface used to access the object

6.34 State diagram

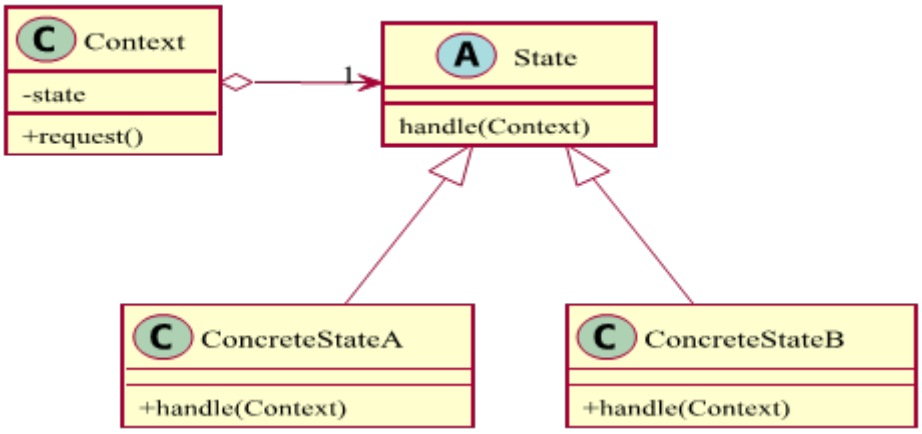


Figure 6.6: State pattern.

6.35 Remarks

- The context holds the concrete state object that provides the behavior according to its current state
- This pattern closely resembles the strategy pattern (upcoming)
- A default behavior has to be defined (i.e. which state to use initially)
- Sometimes one state wants to change itself - this is not possible directly
- Here we could either make the state variable in the Context internal or public (or friend in C++) or return an element of type State

6.36 State sample

```
public class Context
{
    private StateBase _state;

    public Context(StateBase state)
    {
        State = state;
    }

    public void Request()
    {
        _state.Handle(this);
    }

    public StateBase State
    {
        get { return _state; }
```

```
        set { _state = value; }
    }
}

public abstract class StateBase
{
    public abstract void Handle(Context context);
}

class ConcreteStateA : StateBase
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateB();
    }
}

class ConcreteStateB : StateBase
{
    public override void Handle(Context context)
    {
        context.State = new ConcreteStateA();
    }
}
```

6.37 Practical considerations

- Instantiating states can be unnecessary
- Hence buffering state instances might be interesting
- In such cases the factory pattern might be helpful
- States also provide a simple mechanism for extending existing behavior
- We will see that similar patterns, like the template pattern or the strategy patterns exist

6.38 A DVD player

```
public abstract class DVDPlayerState
{
    public abstract DVDPlayerState PlayButtonPressed(DVDPlayer player);

    public abstract DVDPlayerState MenuButtonPressed(DVDPlayer player);
}

public class DVDPlayer
{
    private DVDPlayerState _state;

    public DVDPlayer() : this(new StandbyState())
    {
    }
```

```
    public DVDPlayer(DVDPlayerState state)
    {
        _state = state;
    }

    public void PressPlayButton()
    {
        _state = _state.PlayButtonPressed(this);
    }

    public void PressMenuButton()
    {
        _state = _state.MenuButtonPressed(this);
    }
}

class MenuState : DVDPlayerState
{
    public MenuState()
    {
        Console.WriteLine("MENU");
    }

    public override void PlayButtonPressed(DVDPlayer player)
    {
        Console.WriteLine(" Next Menu Item Selected");
        return this;
    }

    public override void MenuButtonPressed(DVDPlayer player)
    {
        return new StandbyState();
    }
}

class MoviePausedState : DVDPlayerState
{
    public MoviePausedState()
    {
        Console.WriteLine("MOVIE PAUSED");
    }

    public override DVDPlayerState PlayButtonPressed(DVDPlayer player)
    {
        return new MoviePlayingState();
    }

    public override DVDPlayerState MenuButtonPressed(DVDPlayer player)
    {
        return new MenuState();
    }
}

class MoviePlayingState : DVDPlayerState
{
    public MoviePlayingState()
    {
        Console.WriteLine("MOVIE PLAYING");
```

```
    }

    public override DVDPlayerState PlayButtonPressed(DVDPlayer player)
    {
        return new MoviePausedState();
    }

    public override DVDPlayerState MenuButtonPressed(DVDPlayer player)
    {
        return new MenuState();
    }
}

class StandbyState : DVDPlayerState
{
    public StandbyState()
    {
        Console.WriteLine("STANDBY");
    }

    public override DVDPlayerState PlayButtonPressed(DVDPlayer player)
    {
        return new MoviePlayingState();
    }

    public override DVDPlayerState MenuButtonPressed(DVDPlayer player)
    {
        return new MenuState();
    }
}
```

6.39 ... and the strategy pattern

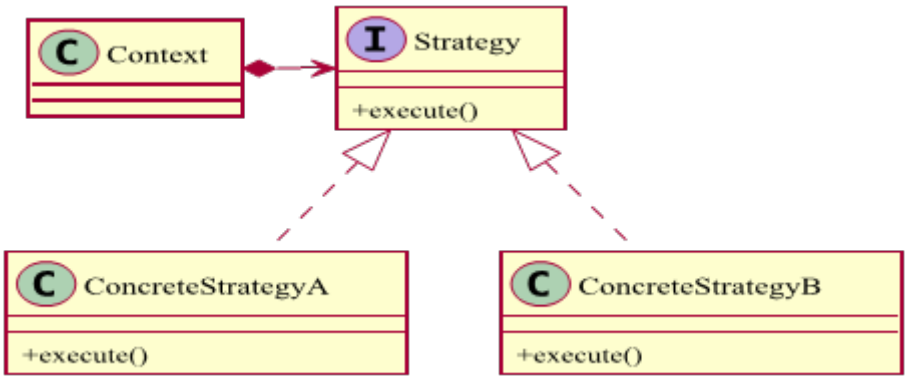


Figure 6.7: Strategy pattern.

6.40 Comparison

- The state pattern is usually quite tightly coupled to a context
- The strategy pattern is completely independent of a context and provides one solution to a problem

- Strategy lets the algorithm vary independently from clients that use it
- A common problem: Choosing one (of many) hashing algorithms
- Solution: An interface that defines the method and various implementations (e.g. HAVAL, MD5, SHA1, SHA2, ...)

6.41 ... and the template pattern

- Quite similar to the state / strategy pattern is the template pattern
- This pattern is based on an abstract class, which defines a set of methods that need to be implemented
- Those methods are then called in a fixed sequence by some client
- While the Strategy design pattern overrides the entire algorithm, this pattern allows us to change only some parts of the behavior algorithm using abstract operations

6.42 Example template sequence

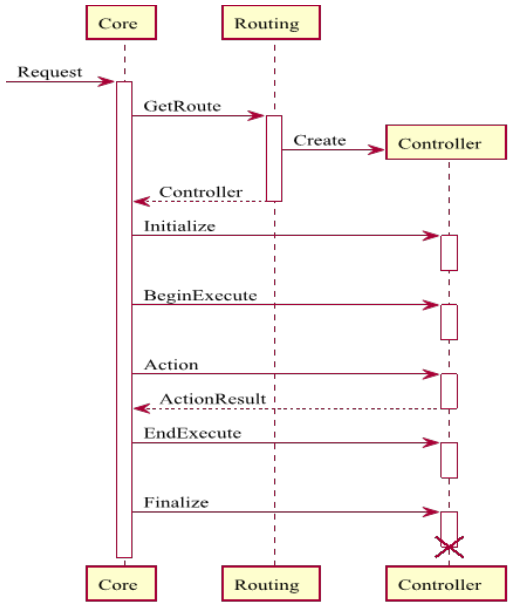


Figure 6.8: Example template pattern sequence.

6.43 Memento pattern

- Saving or restoring states is quite important (e.g. undo, redo)
- The memento pattern defines a memento, a caretaker and an originator
- The originator wants to save or restore his state

- The whole state information is stored in an instance of a class, called memento
- The class responsible for managing the various states is the caretaker

6.44 Memento diagram

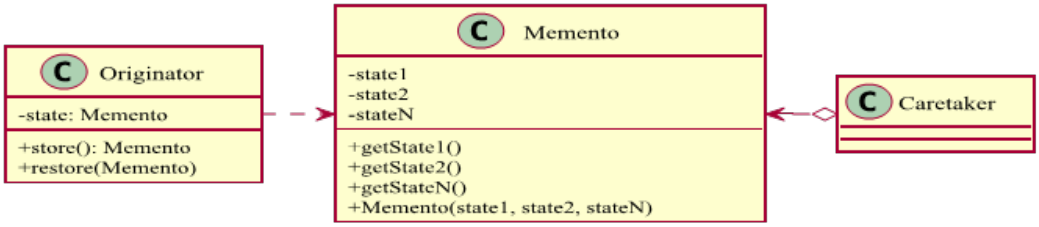


Figure 6.9: Memento pattern.

6.45 Remarks

- The memento is opaque to the caretaker, and the caretaker must not operate on it
- The memento must not allow any operation or access to the internal state
- However, the originator has to access the internal information for restoration
- The memento contains everything that is required for restoring a state

6.46 Memento sample

```
class Originator
{
    private String state;

    public void Set(String state)
    {
        this.state = state;
    }

    public Memento SaveToMemento()
    {
        return new Memento(state);
    }

    public void RestoreFromMemento(Memento memento)
    {
        state = memento.GetSavedState();
    }
}

public class Memento
```

```
{
    private readonly String state;

    public Memento(String stateToSave)
    {
        state = stateToSave;
    }

    public String GetSavedState()
    {
        return state;
    }
}

class Caretaker
{
    List<Memento> savedStates;
    Originator originator;

    public Caretaker()
    {
        savedStates = new List<Memento>();
        originator = new Originator();
    }

    public void ExampleAction()
    {
        originator.Set("State1");
        originator.Set("State2");
        savedStates.Add(originator.SaveToMemento());
        originator.Set("State3");
        savedStates.Add(originator.SaveToMemento());
        originator.Set("State4");
        originator.RestoreFromMemento(savedStates[1]);
    }
}
```

6.47 Specification pattern

- Most applications will be defined by set of rules
- Those rules are usually called business rules or business logic
- The specification pattern allows chaining or recombining objects, which are included in our business logic
- Usually this is build upon three logical operations (and, or, not)
- The classes itself (logic units) will decide if they are applicable
- Chaining is very important for this pattern

6.48 Specification diagram

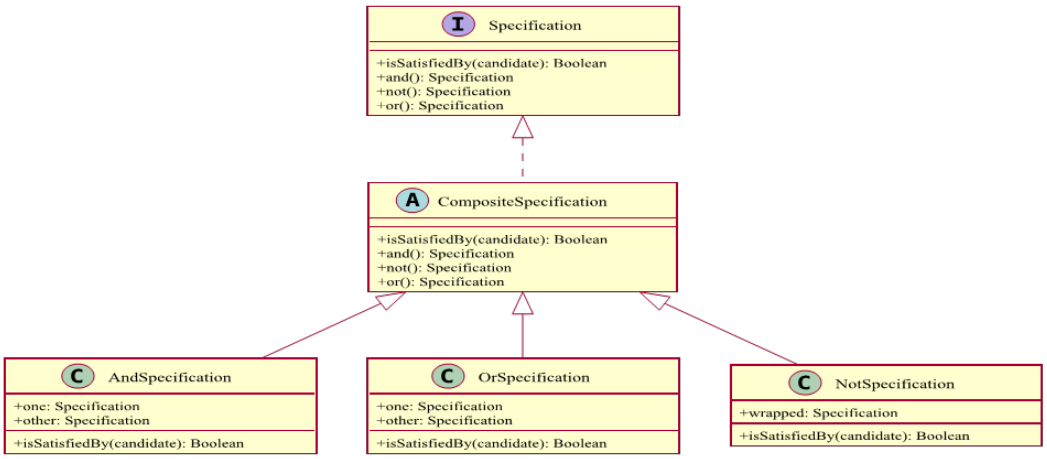


Figure 6.10: Specification pattern.

6.49 Remarks

- The pattern can be used to remove a lot of cruft from a class's interface while decreasing coupling and increasing extensibility
- The primary use is to select a subset of objects based on some criteria
- Languages that allow delegates / function pointers in combination with generics may be already one step ahead
- The pattern's root is in *Domain Driven Design* for criteria tests
- Sometimes data structures are more complex and the visitor pattern might be a useful addition

6.50 Implementation in C#

```
public interface ISpecification<TEntity>
{
    bool IsSatisfiedBy(TEntity entity);
}

internal class AndSpecification<TEntity> : ISpecification<TEntity>
{
    private readonly ISpecification<TEntity> _spec1;
    private readonly ISpecification<TEntity> _spec2;

    protected ISpecification<TEntity> Spec1
    {
        get { return _spec1; }
    }

    protected ISpecification<TEntity> Spec2
```

```
    {
        get { return _spec2; }
    }

    internal AndSpecification(ISpecification<TEntity> spec1, ISpecification<
        TEntity> spec2)
    {
        if (spec1 == null)
            throw new ArgumentNullException("spec1");
        else if (spec2 == null)
            throw new ArgumentNullException("spec2");

        _spec1 = spec1;
        _spec2 = spec2;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) && Spec2.IsSatisfiedBy(candidate);
    }
}

internal class OrSpecification<TEntity> : ISpecification<TEntity>
{
    private readonly ISpecification<TEntity> _spec1;
    private readonly ISpecification<TEntity> _spec2;

    protected ISpecification<TEntity> Spec1
    {
        get { return _spec1; }
    }

    protected ISpecification<TEntity> Spec2
    {
        get { return _spec2; }
    }

    internal OrSpecification(ISpecification<TEntity> spec1, ISpecification<
        TEntity> spec2)
    {
        if (spec1 == null)
            throw new ArgumentNullException("spec1");
        else if (spec2 == null)
            throw new ArgumentNullException("spec2");

        _spec1 = spec1;
        _spec2 = spec2;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return Spec1.IsSatisfiedBy(candidate) || Spec2.IsSatisfiedBy(candidate);
    }
}

internal class NotSpecification<TEntity> : ISpecification<TEntity>
{
    private readonly ISpecification<TEntity> _wrapped;
```

```
    protected ISpecification<TEntity> Wrapped
    {
        get { return _wrapped; }
    }

    internal NotSpecification(ISpecification<TEntity> spec)
    {
        if (spec == null)
            throw new ArgumentNullException("spec");

        _wrapped = spec;
    }

    public bool IsSatisfiedBy(TEntity candidate)
    {
        return !Wrapped.IsSatisfiedBy(candidate);
    }
}

public static class ExtensionMethods
{
    public static ISpecification<TEntity> And<TEntity>(this ISpecification<
        TEntity> spec1, ISpecification<TEntity> spec2)
    {
        return new AndSpecification<TEntity>(spec1, spec2);
    }

    public static ISpecification<TEntity> Or<TEntity>(this ISpecification<
        TEntity> spec1, ISpecification<TEntity> spec2)
    {
        return new OrSpecification<TEntity>(spec1, spec2);
    }

    public static ISpecification<TEntity> Not<TEntity>(this ISpecification<
        TEntity> spec)
    {
        return new NotSpecification<TEntity>(spec);
    }
}
```

6.51 Mediator pattern

- To avoid strong coupling between two classes we can introduce a mediator class
- The mediator knows the two classes and establishes the connection
- Basically it is a reduction of complexity
- The reduction makes sense if both classes are quite heavy
- In most cases using a simple interface instead can be even better

6.52 Mediator diagram

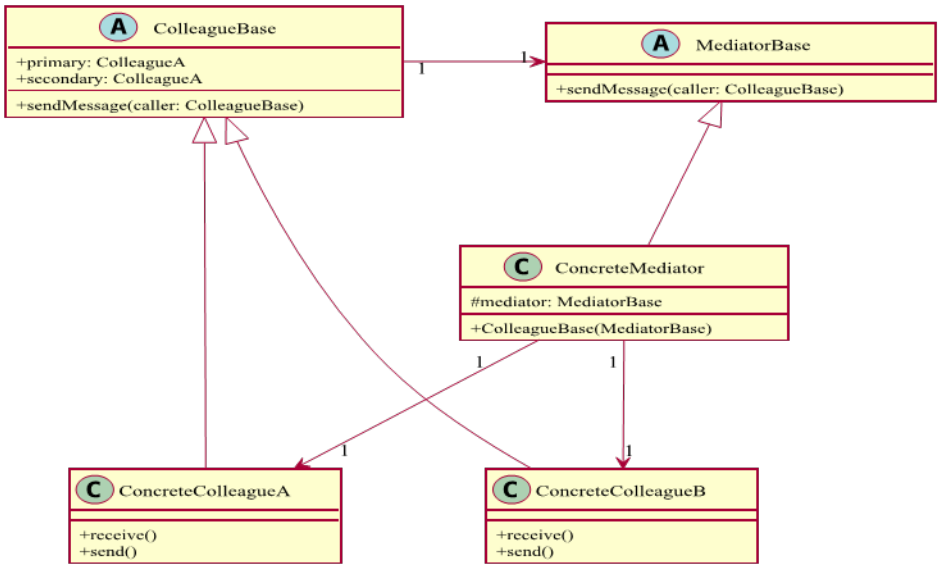


Figure 6.11: Mediator pattern

6.53 Remarks

- Usually the mediator tries to be transparent
- For testability an interface is the best solution (mocking)
- Using a middle object decouples two classes
- We will see that this idea is the basis for more advanced techniques
- Basis: Reflection and compile-time decoupling by run-time wiring
- Dependency Injection uses a class that talks to all other classes
- The concrete classes do not have to be known

6.54 Two-way mediator sequence

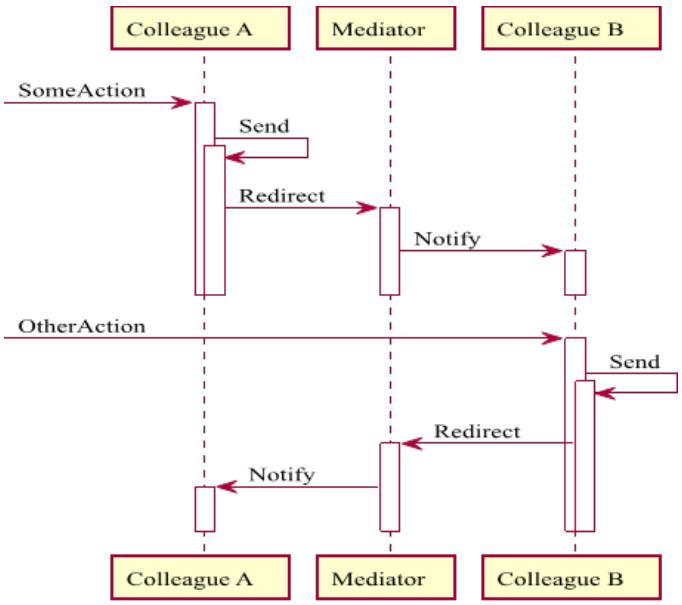


Figure 6.12: Mediator sequences.

6.55 Interpreter pattern

- Even simple applications might require some kind of DSL
- A DSL (Domain-Specific Language) is a kind of language for a special kind of application
- This is in contrast to GPL (general purpose languages)
- There are various types of DSLs like markup, modeling or programming languages
- Arguments of a command line might be already dynamic enough to define a DSL specification

6.56 How does the interpreter pattern help?

- The interpreter pattern allows the grammar for such a DSL to be represented as classes
- This (object-oriented) representation can then easily be extended
- The whole process takes information from a *Context*
- A *Client* creates the context and starts the interpreter by calling the *interpret* action
- This action is located on an object of type *Expression*

6.57 Interpreter diagram

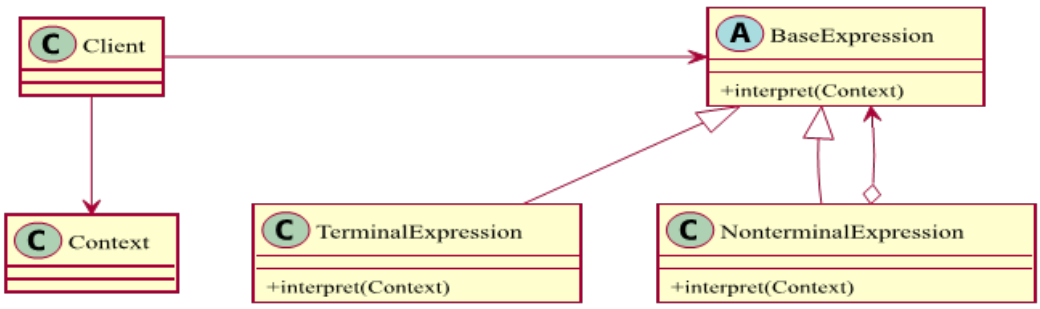


Figure 6.13: Interpreter pattern.

6.58 Remarks

- The context contains information that is global to the interpreter
- The expressions represent the units of grammar
- The state pattern can be quite useful for parsing
- *NonterminalExpression* expressions are aggregates containing one or more (further) expressions (conditionally chained)
- The *TerminalExpression* implements an interpret operation associated with terminal symbols in the grammar
- An instance is required for every terminal symbol in the given string

6.59 An interpreter for roman numbers

```
class Client
{
  public static int Parse(string number)
  {
    Context context = new Context(number);
    List<Expression> tree = new List<Expression>();
    tree.Add(new ThousandExpression());
    tree.Add(new HundredExpression());
    tree.Add(new TenExpression());
    tree.Add(new OneExpression());

    foreach (Expression exp in tree)
      exp.Interpret(context);

    return context.Value;
  }
}

class Context
{
```

```
  private string query;
  private int value;

  public Context(string input)
  {
    query = input;
  }

  public string Query
  {
    get { return query; }
    set { this.query = value; }
  }

  public int Value
  {
    get { return value; }
    set { this.value = value; }
  }
}

abstract class Expression
{
    public void Interpret(Context context)
    {
    if (context.Query.Length == 0)
      return;

    if (context.Query.StartsWith(Nine))
    {
      context.Value += 9 * Weight;
      context.Query = context.Query.Substring(Nine.Length);
    }
    else if (context.Query.StartsWith(Four))
    {
      context.Value += 4 * Weight;
      context.Query = context.Query.Substring(Four.Length);
    }
    else if (context.Query.StartsWith(Five))
    {
      context.Value += 5 * Weight;
      context.Query = context.Query.Substring(Five.Length);
    }

    while (context.Query.StartsWith(One))
    {
      context.Value += Weight;
      context.Query = context.Query.Substring(One.Length);
    }
    }

    public virtual string One { get { return " "; } }

    public virtual string Four { get { return " "; } }

    public virtual string Five { get { return " "; } }

    public virtual string Nine { get { return " "; } }
```

```
    public abstract int Weight { get; }
}

class ThousandExpression : Expression
{
  public override string One { get { return "M"; } }

  public override int Weight { get { return 1000; } }
}

class HundredExpression : Expression
{
  public override string One { get { return "C"; } }

  public override string Four { get { return "CD"; } }

  public override string Five { get { return "D"; } }

  public override string Nine { get { return "CM"; } }

  public override int Weight { get { return 100; } }
}

class TenExpression : Expression
{
  public override string One { get { return "X"; } }

  public override string Four { get { return "XL"; } }

  public override string Five { get { return "L"; } }

  public override string Nine { get { return "XC"; } }

  public override int Weight { get { return 10; } }
}

class OneExpression : Expression
{
  public override string One { get { return "I"; } }

  public override string Four { get { return "IV"; } }

  public override string Five { get { return "V"; } }

  public override string Nine { get { return "IX"; } }

  public override int Weight { get { return 1; } }
}
```

6.60 Practical considerations

- Basic idea: A class for each defined symbol
- The structure of the syntax tree is given by the composite pattern
- Compilers are usually not that simple, since they are more fine-grained

- Simple languages like SQL are, however, perfect suitable
- In practice the client is called the engine and invoked from a real client
- The client usually already contains some logic

6.61 References

- CodeProject: Design Patterns 3 of 3 (`http://www.codeproject.com/Articles/455228/Design-Patterns-3-of-3-Behavioral-Design-Patterns`)
- CodeProject: The Strategy Pattern (`http://www.codeproject.com/Articles/52807/Design-Patterns-Part-1-The-Strategy-Pattern`)
- Wikipedia: Specification pattern (`http://en.wikipedia.org/wiki/Specification_pattern`)
- Wikipedia: Interpreter pattern (`http://en.wikipedia.org/wiki/Interpreter_pattern`)
- Dofactory: Pattern overview (`http://www.dofactory.com/Patterns/Patterns.aspx`)
- Iterator pattern demystified (`http://www.codeproject.com/Tips/359873/Csharp-Iterator-Pattern-demystified`)

6.62 Literature

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object Oriented Software.*
- Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). *Head First Design Patterns.*
- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.*
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture.*

Chapter 7

Structural patterns

7.1 Introduction

- Deal with the structure of a program
- Try to ease the design by identifying ways to realize certain relationships
- They are concerned with how classes and objects are composed to form larger structures
- Goal: Clean big picture, with a better (reliable, extensible, ...) layout
- Structural *class* patterns use inheritance to compose interfaces or implementations

7.2 The adapter pattern

- Old problem: One class wants to talk to another class by using a specific method, which is named differently
- Solution in the real world: An adapter is required!
- The adapter pattern tries to allow communication between two incompatible types
- The central class is called the *Adapter*
- This class knows about the *Adaptee* and the specific *Target* that is required from a client

7.3 Adapter diagram

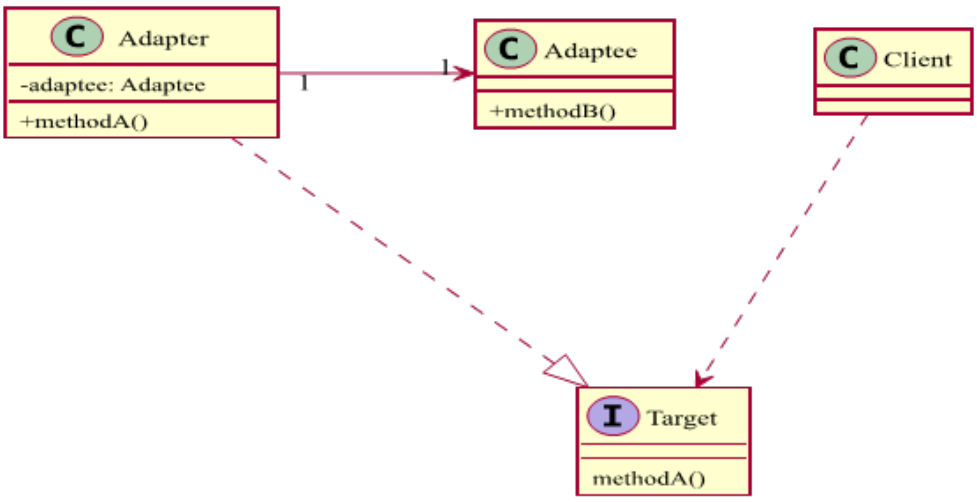


Figure 7.1: Adapter pattern.

7.4 Remarks

- The adapter pattern is always required when we want to enable communication between two boxed systems
- Sometimes this is also known as **Wrapper**
- It can create a reusable class that cooperates with unrelated classes that have incompatible interfaces
- One-way, two-way? Classically only a one-way solution is supported
- Two-way requires interfaces (Java, C#) or multiple-inheritance (C++)

7.5 A simple adapter

```
public interface ITarget
{
    void MethodA();
}
public class Client
{
    private readonly ITarget _target;

    public Client(ITarget target)
    {
        _target = target;
    }

    public void Request()
    {
```

```
        _target.MethodA();
    }
}
public class Adaptee
{
    public void MethodB()
    {
        /* ... */
    }
}
public class Adapter : ITarget
{
    private readonly Adaptee _adaptee = new Adaptee();

    public void MethodA()
    {
        _adaptee.MethodB();
    }
}
```

7.6 Practical considerations

- If the class inheriting from one class has a reference to the other class we call it an *aggregate*
- Otherwise if we do only method renaming for generating compatibility we call it a *compatible*
- In practice the adapter pattern is also used within the same library
- Some of the upcoming patterns are also helpful for generating a common communication platform

7.7 The proxy pattern

- A proxy is an object that performs transparent communication by tunneling through a system
- Usually we are using proxies to hide the real data / implementation
- A proxy is a class with methods, which has a reference to another class with data
- Only the proxy can modify the data or execute methods of the data class
- This has security advantages, however, only at compiler level (not RT)

7.8 Proxy diagram

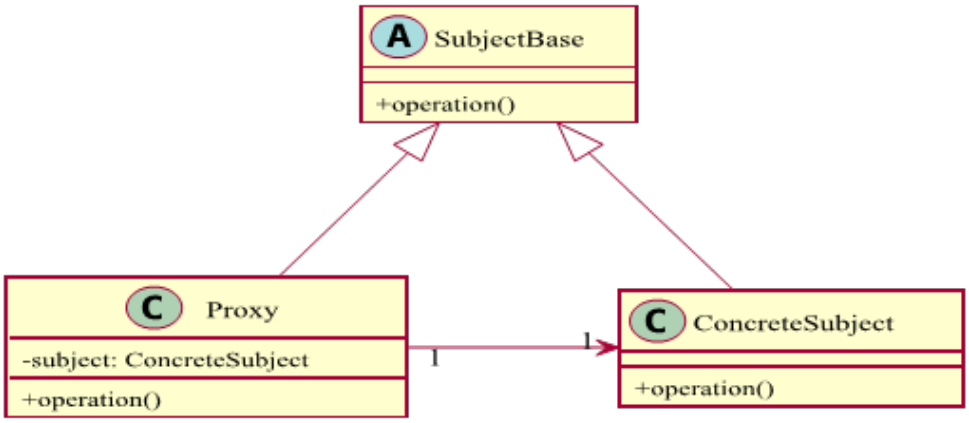


Figure 7.2: Proxy pattern.

7.9 Remarks

- Proxies enable indirect access to data
- We have security and memory advantages
- The data has handled only by one class
- The data is encapsulated and passed by reference
- The abstraction is usually contained in an interface
- Implementation-wise an abstract class could be better (nested classes can access private members of parent class objects)

7.10 A transparent wrapper

```
public abstract class SubjectBase
{
    public abstract void Operation();
}
public class ConcreteSubject : SubjectBase
{
    public override void Operation()
    {
        /* ... */
    }
}
public class Proxy : SubjectBase
{
    private ConcreteSubject subject;

    public override void Operation()
    {
```

```
        if (subject == null)
            subject = new ConcreteSubject();

        subject.Operation();
    }
}
```

7.11 Practical considerations

- Usually the proxy pattern is a way to minimize data duplication
- This is then a special case of the flyweight pattern (upcoming)
- Sometimes proxies come with a static instance counter
- Multiple proxies increase the instance counter and use the same data object
- Once no instances are left, the data object is disposed
- The reference counter pointer object is an implementation of this pattern

7.12 The bridge pattern

- A bridge allows us to reach the other side of a river or valley
- Here we connect to an object of a certain type
- The specific object could be changed
- Therefore we separate the abstraction from the implementation
- Changes in the implementation do not affect the abstraction
- This is very helpful as seen with the state pattern

7.13 Bridge diagram

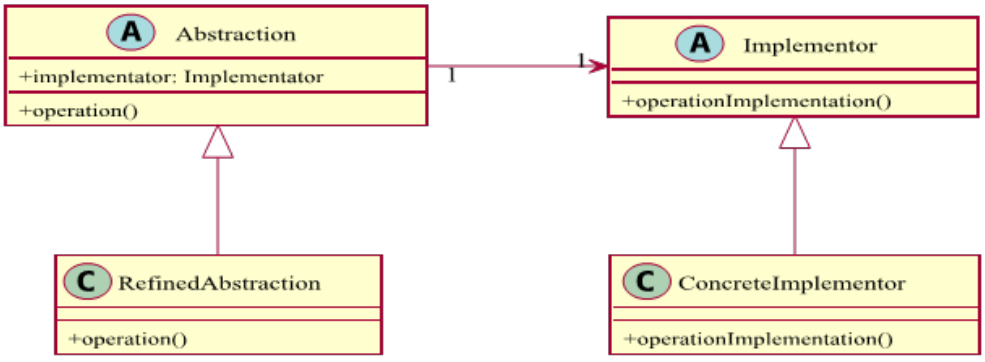


Figure 7.3: Bridge pattern.

7.14 Remarks

- The bridge pattern provides a cleaner implementation of real-world objects
- The implementation details can be changed easily
- Additionally it is possible to consider a variation of implementations
- Also the bridge pattern decouples the usage of a concrete implementation with a client, which ensures testability and robustness

7.15 A simple abstraction layer

```
abstract class Implementor
{
    public abstract void Execute();
}
class Abstraction
{
    protected Implementor implementor;

    public Implementor Implementor
    {
        get { return implementor; }
        set { implementor = value; }
    }

    public virtual void Operation()
    {
        implementor.Execute();
    }
}
class RefinedAbstraction : Abstraction
{
    public override void Operation()
    {
        implementor.Execute();
    }
}
class ConcreteImplementor : Implementor
{
    public override void Execute()
    {
        /* ... */
    }
}
```

7.16 Practical considerations

- Practically the bridge pattern is used quite often with interfaces
- Interfaces already bring in some advantages like testability (*mocking*)
- If the implementation is defined by an interface, then the bridge does not contain any unnecessary overhead

- The proxy might also be responsible for handling the resources
- This pattern is useful to share an implementation among multiple objects

7.17 The facade pattern

- A facade is a nice wall that hides the underlying building
- Usually a facade is used to provide the API behind a complex system
- Also it might be useful to bundle dependencies on external libraries
- This reduces dependencies for other packages
- A facade can create a new API that calls the required methods of the other APIs
- Hence a facade bundles different classes to achieve a common goal

7.18 Facade diagram

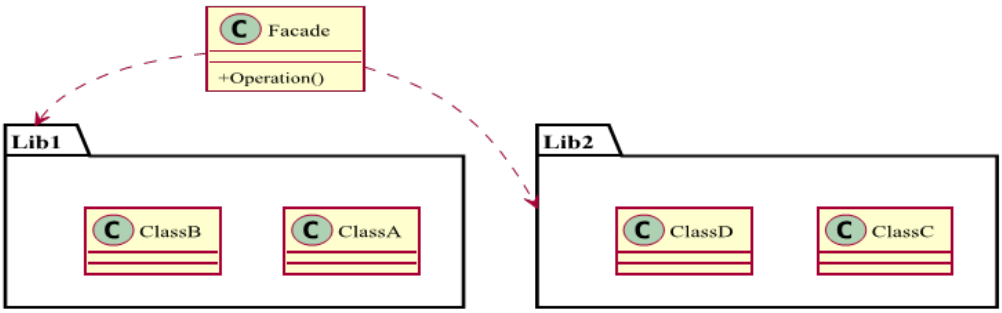


Figure 7.4: Facade pattern.

7.19 Remarks

- Classically we have one top class that contains references to each class's instance
- Each class represents a set of subtasks
- The facade's methods use several methods of the contained instances
- This constructs a save way of communicating to instances that do not share a common basis, but a common goal
- Testing an API by only testing the top-level increases testability
- The facade pattern also enhances readability

7.20 Wrapping libraries

```
public class Facade
{
    public void PerformAction()
    {
        var c1a = new Class1A();
        var c1b = new Class1B();
        var c2a = new Class2A();
        var c2b = new Class2B();
        var result1a = c1a.Func();
        var result1b = c1b.Func(result1a);
        var result2a = c2a.Func(result1a);
        c2b.Action(result1b, result2a);
    }
}
public class Class1A
{
    public int Func()
    {
        Console.WriteLine("Class1A.Func return value: 1");
        return 1;
    }
}
public class Class1B
{
    public int Func(int param)
    {
        Console.WriteLine("Class1B.Func return value: {0}", param + 1);
        return param+1;
    }
}
public class Class2A
{
    public int Func(int param)
    {
        Console.WriteLine("Class2A.Func return value: {0}", param + 2);
        return param+2;
    }
}
public class Class2B
{
    public void Action(int param1, int param2)
    {
        Console.WriteLine("Class2B.Action received: {0}", param1 + param2);
    }
}
```

7.21 Practical considerations

- Quite often poorly designed APIs are also wrapped with a facade
- A facade is used when one wants an easier or simpler interface to an underlying implementation object
- The difference to an adapter is that an adapter also respects a particular interface and supports polymorphic behavior

- A facade does not support polymorphic behavior
- Here a decorator (upcoming) should be used for extensions

7.22 The flyweight pattern

- Several objects containing unique data and shared data
- A static class containing the shared data objects (or *Singleton*)
- A class to represent the shared data
- This should reduce the memory footprint
- Reducing the memory requirement will increase the performance
- In certain cases this can improve consistency

7.23 Flyweight diagram

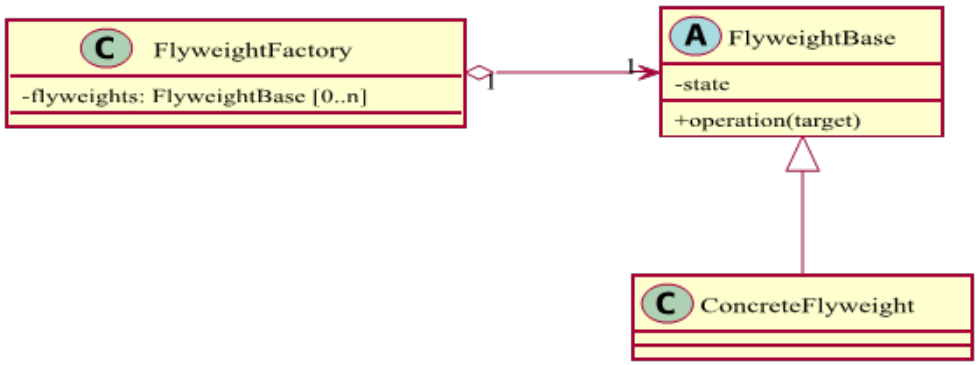


Figure 7.5: Flyweight pattern.

7.24 Remarks

- This pattern makes most sense when a huge amount of data is considered or a collection of immutable objects
- The objects should either not change at all or be shared among several clients
- An application is the string management in languages like Java or C#
- Here every (unique) string is stored in a string table
- Most strings therefore do not need to be instantiated, as they are available in the table (otherwise they are created and added)

7.25 Flyweight factory

```
public class FlyweightFactory
{
    private readonly Dictionary<string, FlyweightBase> _flyweights;

    public FlyweightFactory()
    {
        _flyweights = new Dictionary<string, FlyweightBase>();
    }

    public FlyweightBase GetFlyweight(string key)
    {
        if (_flyweights.ContainsKey(key))
        {
            return _flyweights[key];
        }
        else
        {
            var newFlyweight = new ConcreteFlyweight();
            _flyweights.Add(key, newFlyweight);
            return newFlyweight;
        }
    }
}
```

7.26 Practical considerations

- The factory might follow the factory pattern
- However, instead of instantiating objects we just return them
- In some cases (lazy loading) we might return special objects ...
- ... or instantiate objects if certain criteria are met
- Sometimes it is useful to include resource management
- A perfect example is a StringBuilder pool
- Here we can gain a lot of performance for creating (long) strings on multiple occasions

7.27 Sharing states

```
public Glyph
{
    private int width;
    private int height;
    private int ascent;
    private int descent;
    private int pointSize;

    public int Width
    {
```

```
      get { return width; }
    }

    public int Height
    {
      get { return height; }
    }

    public int Ascent
    {
      get { return ascent; }
    }

    public int Descent
    {
      get { return descent; }
    }

    public int PointSize
    {
      get { return pointSize; }
    }
}
public Character
{
  char letter;
  int position;
  Glyph properties;

  public Character(char letter, int position)
  {
    this.letter = letter;
    this.position = position;
    this.properties = GlyphFactory.Get(letter);
  }

  public char Letter
  {
    get { return letter; }
  }

  public int Position
  {
    get { return position; }
  }

  public Glyph Properties
  {
    get { return properties; }
  }
}
```

7.28 The decorator pattern

- Quite often we have to extend existing classes
- Sometimes however, these classes are sealed or cannot be extended directly

- A pattern that is helpful in such cases is the decorator pattern
- Here we need a decorator class that implements the same interface as the abstraction
- This decorator then requires an object that implements the interface as well

7.29 Decorator diagram

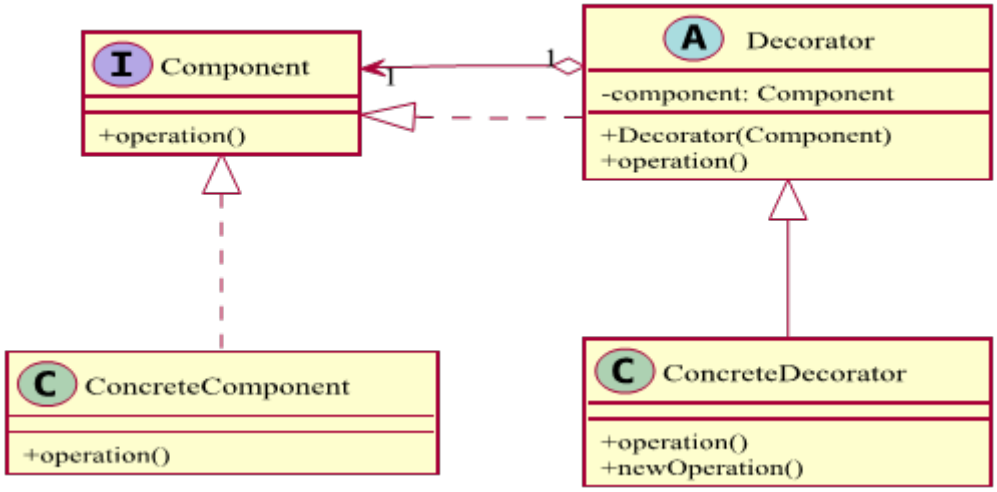


Figure 7.6: Decorator pattern.

7.30 Remarks

- There might be classes inheriting from the decorator
- This concept enables more sophisticated ways of using the interface
- Actually this pattern is quite close to the builder pattern
- However, the decorator is more focused on the model itself than on modifying it / general behavior
- It decouples complexity by using inheritance to specialize objects

7.31 Decorating existing classes

```
public interface IComponent
{
    void Operation();
}
public class ConcreteComponent : IComponent
{
    public void Operation()
```

```
    {
        /* ... */
    }
}
public class Decorator : IComponent
{
    private readonly IComponent _component;

    public Decorator(IComponent component)
    {
        _component = component;
    }

    public void Operation()
    {
        _component.Operation();
    }
}
```

7.32 Practical considerations

- The decorater pattern works best if the component is an interface
- In principle this is also a wrapper, however, a non-transparent one
- This wrapper might also extend the existing interface, by e.g. adding new operations or properties
- The big advantage is that this wrapper is pluggable, i.e. it can be used with any object of the given time at runtime
- A decorator could also be used as input for a decorator (stacking)

7.33 Sandwiches

```
public abstract class Sandwich
{
    private string description;

    public abstract double Price { get; }

    public virtual string Description
    {
        get { return description; }
        protected set { description = value; }
    }
}
public class TunaSandwich : Sandwich
{
    public TunaSandwich()
    {
        Description = "Tuna Sandwich";
    }

    public override double Price
```

```
    {
        get { return 4.10; }
    }
}
public class SandwichDecorator : Sandwich
{
    protected Sandwich sandwich;
    private string description;

    public SandwichDecorator(Sandwich sandwich)
    {
        this.sandwich = sandwich;
    }

    public override string Description
    {
        get { return sandwich.Description + ", " + description; }
        protected set { description = value; }
    }
    public override double Price
    {
        get { return sandwich.Price; }
    }
}
public class Cheese : SandwichDecorator
{
    public Cheese(Sandwich sandwich) : base(sandwich)
    {
        Description = "Cheese";
    }
    public override double Price
    {
        get { return sandwich.Price + 1.23; }
    }
}
```

7.34 The composite pattern

- How to efficiently create tree-like structures?
- The answer is to follow the composite pattern
- One object containing several other objects
- We have two different kind of objects in there, leafs (certainly without child nodes) and composites (possibly containing child nodes)
- However, both types are subtypes of a *Component* class
- Most UI systems follow the composite pattern

7.35 Composite diagram

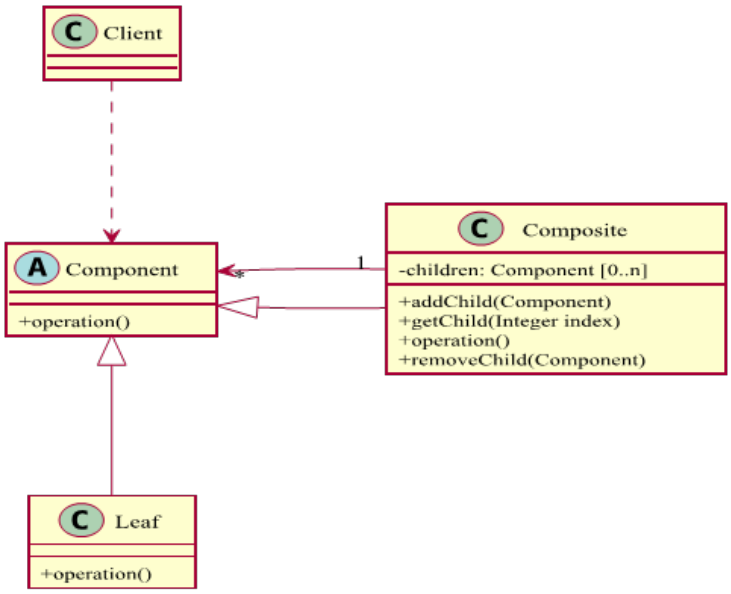


Figure 7.7: Composite pattern.

7.36 Remarks

- Ideally the component defines the methods that are shared among all nodes
- Usually it makes sense to have this component defined as an interface
- The interface should have a method to allow enumeration
- The enumeration lists all children and their children
- This results in a recursive call at each node level
- Less redundancy and formal more correct than other approaches

7.37 Building a tree structure

```
public abstract class Component
{
    protected readonly string name;

    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Operation();
    public abstract void Show();
}
class Composite : Component
```

```
{
    private readonly List<Component> _children;

    public Composite(string name)
        : base(name)
    {
        _children = new List<Component>();
    }

    public void AddChild(Component component)
    {
        _children.Add(component);
    }
    public void RemoveChild(Component component)
    {
        _children.Remove(component);
    }
    public Component GetChild(int index)
    {
        return _children[index];
    }
    public override void Operation()
    {
        Console.WriteLine("Composite with " + _children.Count + " child(ren).");
    }
    public override void Show()
    {
        Console.WriteLine(name);

        foreach (Component component in _children)
        {
            component.Show();
        }
    }
}
public class Leaf : Component
{
    public Leaf(string name)
        : base(name)
    {
    }

    public override void Operation()
    {
        Console.WriteLine("Leaf.");
    }
    public override void Show()
    {
        Console.WriteLine(name);
    }
}
```

7.38 Snapshot

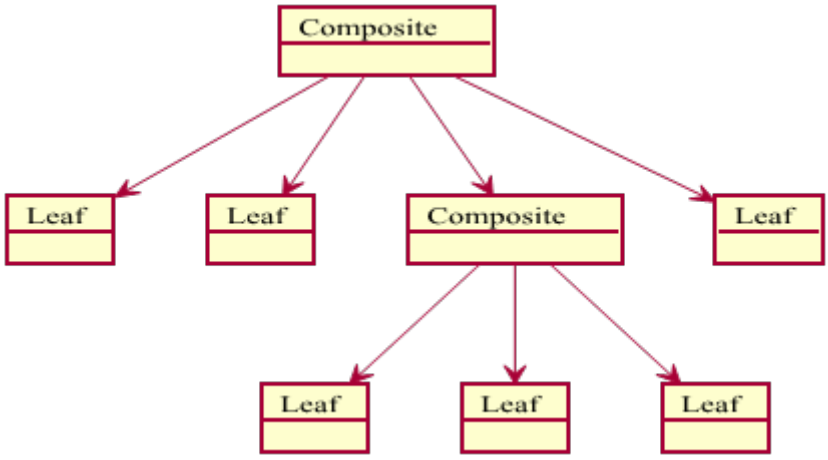


Figure 7.8: A sample composite object structure.

7.39 The aggregate pattern

- Quite similar to the composite pattern is the aggregate pattern
- This pattern has emerged from *Domain-Driven Design* (DDD)
- Instead of treating objects as individuals we consider a single item
- One object is therefore called the *aggregate root*, which handles all calls
- The root ensures the integrity of the aggregate as a whole
- These aggregates should not be confused with collections
- Collections are generic, aggregates are specialized and might contain additional fields or multiple collections

7.40 Differences

- The aggregate pattern is based on composition
- The composite pattern is based on aggregation
- That being said, it should be obvious that the constructor of an aggregate requires the root object, since its existence is bound to a root
- An aggregate consists of 1..n objects
- A composite consists of 0..n objects
- The aggregate is destroyed when the root object is disposed

7.41 Practical considerations

- An explicit parent reference simplifies moving up the tree
- It makes sense to define the parent inside the component class
- Components might be shared unless more than one parent is possible
- If the number of children is small it can make sense to put the list with children inside the component class
- Caching might be useful if searching for children is expensive
- The composite class might contain any data structure from linked lists to trees, arrays and hash tables

7.42 References

- CodeProject: Design Patterns 2 of 3 (`http://www.codeproject.com/Articles/438922/Design-Patterns-2-of-3-Structural-Design-Patterns`)
- CodeProject: Illustrated GOF Design Patterns (Part II) (`http://www.codeproject.com/Articles/3151/Illustrated-GOF-Design-Patterns-in-C-Part-II-Struc`)
- Wikipedia: Structural pattern (`http://en.wikipedia.org/wiki/Structural_pattern`)
- Dofactory: Pattern overview (`http://www.dofactory.com/Patterns/Patterns.aspx`)
- Sourcemaking: The Flyweight pattern (`http://sourcemaking.com/design_patterns/flyweight`)
- Silversoft about structural patterns (`http://www.silversoft.net/docs/dp/hires/chap4fs.htm`)

7.43 Literature

- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object Oriented Software.*
- Freeman, Eric; Freeman, Elisabeth; Kathy, Sierra; Bert, Bates (2004). *Head First Design Patterns.*
- Hannemann, Jan (2002). *Design pattern implementation in Java and AspectJ.*
- Fowler, Martin (2006). *Writing Software Patterns.*
- Liskov, Barbara; Guttag, John (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.*

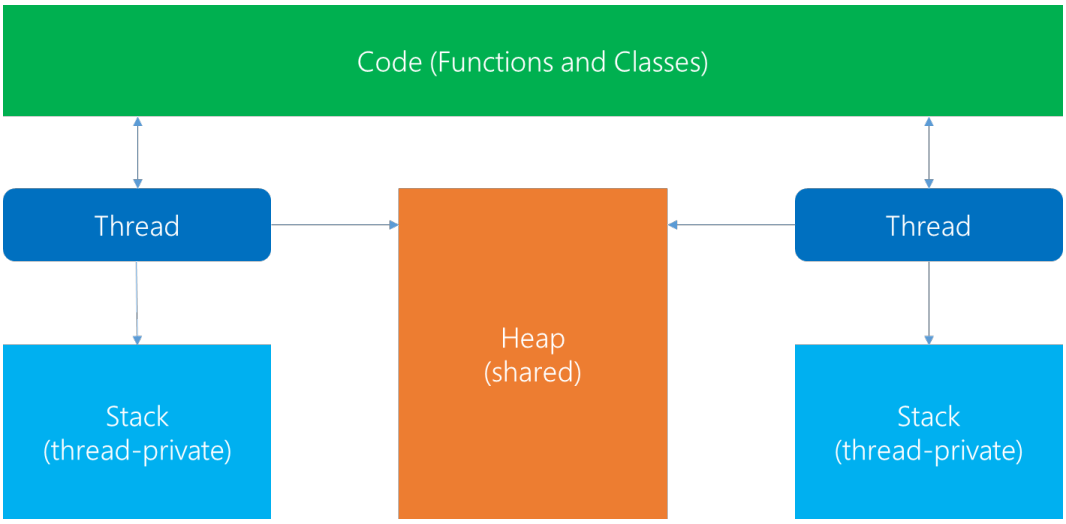
Chapter 8

Concurrency patterns

8.1 Introduction

- They deal with the multi-threaded programming paradigm
- However, in general they describe asynchronous processes
- This does not necessary mean multi-threaded
- Usually one does not need threads for async behavior
- Goal: Reduce communication overhead, while increasing flexibility
- Manage program resources more efficiently

8.2 Threading overview



8.3 Problems and primitives

- Multi-threading might end in race conditions

- A race-condition is when two or more threads want to access a specific resource at the same time
- Also memory inconsistencies like instruction re-ordering or only partially constructed objects might arise
- To cover this we can use *Mutex* objects
- These are mutually exclusive and the basis for e.g. barriers, ...

8.4 The scheduler pattern

- Old problem: How to synchronize access?
- Special case: Limit access to a specific resource to a single thread
- The result is that the resource will always be accessed synchronized
- Use-case: Synchronized access to the elements in a GUI
- The scheduler pattern provides the basis for sequencing waiting threads
- It is similar to the mediator pattern, with the various threads being mediated

8.5 Scheduler diagram

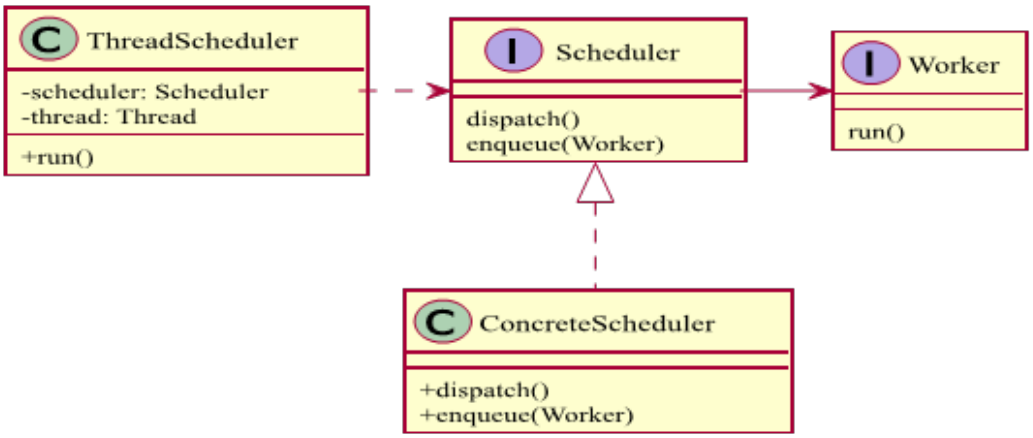


Figure 8.1: Scheduler pattern.

8.6 Remarks

- Usually a mechanism to implement a scheduling policy is provided
- This mechanism is defined in an abstract class
- The pattern is independent of a specific scheduling policy
- This policy might be FIFO, FILO, priority queue, ... or customized

- This pattern adds some overhead beyond a barrier or similar locking techniques

8.7 Sample implementation

```
interface IWorker
{
  void Run();
}

interface IScheduler
{
  void Dispatch();
  void Enqueue(IWorker worker);
}

class ThreadScheduler
{
  IScheduler scheduler;
  Thread thread;

  public ThreadScheduler(IScheduler scheduler)
  {
    this.scheduler = scheduler;
    this.thread = new Thread(Loop);
  }

  public void Run()
  {
    thread.Start();
  }

  void Loop()
  {
    while (true)
    {
      scheduler.Dispatch();
      Thread.Sleep(10);
    }
  }
}

class QueueScheduler : IScheduler
{
  Queue<IWorker> queue;
  Object key;

  public QueueScheduler()
  {
    queue = new Queue<IWorker>();
    key = new Object();
  }

  public void Dispatch()
  {
    IWorker process;
```

```
    lock (key)
    {
      if (queue.Count == 0)
        return;

      process = queue.Dequeue();
    }

    process.Run();
  }

  public void Enqueue(IWorker worker)
  {
    lock (key)
    {
      queue.Enqueue(worker);
    }
  }
}

class SleepyWorker : IWorker
{
  public void Run()
  {
    Console.WriteLine("Sleeping 1s...");
    Thread.Sleep(1000);
    Console.WriteLine("Finished sleeping!");
  }
}

class WorkingWorker : IWorker
{
  public void Run()
  {
    Console.WriteLine("Doing hard work for 2s...");
    Thread.Sleep(2000);
    Console.WriteLine("Finished doing work!");
  }
}
```

8.8 Practical considerations

- The read/write lock pattern can be implemented using the scheduler pattern
- The .NET TPL uses this pattern to provide a scheduler for tasks
- Also Intel TBB offers a task scheduler, which can also be used to accommodate a given worker size
- In general schedulers might be used for various scenarios, not only if a certain task needs to be executed single-threaded

8.9 The monitor pattern

- Synchronization is important and sometimes a certain condition has to be fulfilled for allowing thread(s) to continue
- The monitor object pattern allows both:
  1. mutual exclusion
  2. ability to wait for a certain condition
- Therefore this pattern is thread-safe and condition-driven
- Additionally we might want to inform a thread about condition-changes

8.10 Monitor diagram

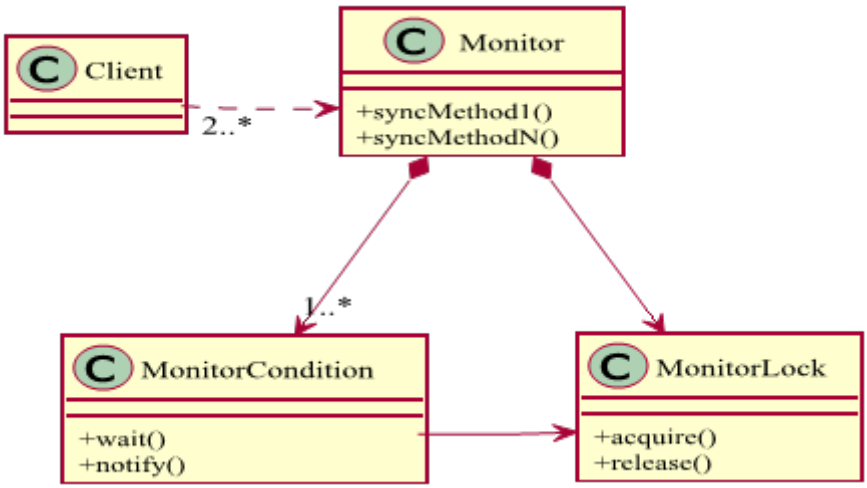


Figure 8.2: Monitor pattern.

8.11 Remarks

- A condition variable consists of threads that are waiting on a certain condition to be fulfilled
- Monitors provide a mechanism for threads to temporarily give up exclusive access
- This is done in order to wait for some condition to be met
- After the condition is met the exclusive access is regained
- Their task is then continued (threads are resumed)

8.12 Sample implementation

```
class MonitorLock
{
}

class MonitorCondition
{
  MonitorLock mlock;
  Queue<Thread> threads;

  public MonitorCondition(MonitorLock _lock)
  {
    threads = new Queue<Thread>();
    mlock = _lock;
  }

  public void Wait()
  {
    var willSleep = false;

    lock (mlock)
    {
      willSleep = threads.Count > 0;
      threads.Enqueue(Thread.CurrentThread);
    }

    if (willSleep)
    {
      try { Thread.Sleep(Timeout.Infinite); }
      catch (ThreadInterruptedException) { }
    }
  }

  public void Notify()
  {
    var willInterrupt = false;

    lock (mlock)
    {
      willInterrupt = threads.Count > 0;
      threads.Dequeue();
    }

    if (willInterrupt)
      threads.Peek().Interrupt();
  }
}

class Monitor
{
  MonitorLock mlock;
  MonitorCondition mcondition;

  public Monitor()
  {
    mlock = new MonitorLock();
    mcondition = new MonitorCondition(mlock);
```

```
  }

  public void Tick()
  {
    mcondition.Wait();
    Thread.Sleep(1000);
    Console.WriteLine("Tick");
    mcondition.Notify();
  }

  public void Tock()
  {
    mcondition.Wait();
    Thread.Sleep(1000);
    Console.WriteLine("Tock");
    mcondition.Notify();
  }
}
```

8.13 Practical considerations

- Each condition represents a waiting room
- The *Notify* method is used to wake up a waiting process
- In real-life this works like a fast food restaurant
- Separate concerns and protect object state from uncontrolled changes
- Objects should be responsible for ensuring that any of their methods that require synchronization are serialized transparently
- Conditions act as a scheduler (in fact a scheduler might be considered)
- Be aware of the nested monitor lockout (**no** nesting!)

8.14 Sample monitor sequence

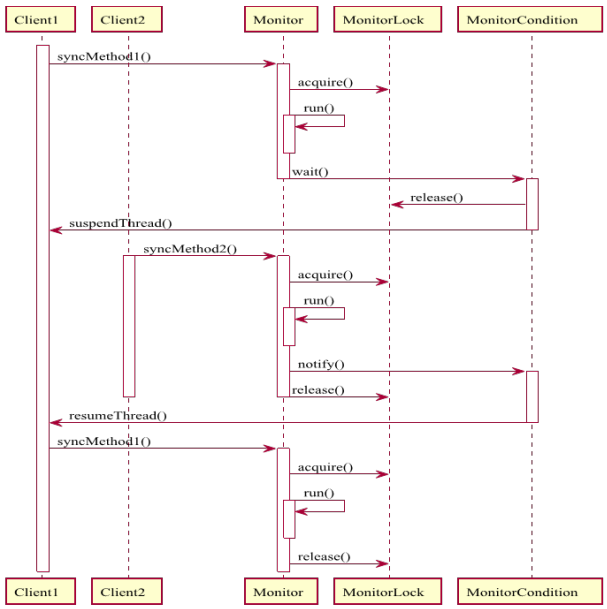


Figure 8.3: Monitor Object sample.

8.15 The thread pool pattern

- Usually we don't want to create an arbitrary number of threads
- Instead we want a maximum number of threads to handle open tasks
- The solution is to use a thread pool, which recycles and limits threads
- Idea: Obtain threads faster, reduce used resources, optimal usage
- A thread pool consists of a task queue and a pool of running workers
- If a thread pool is available we should definitely prefer it to plain threads

8.16 Thread pool diagram

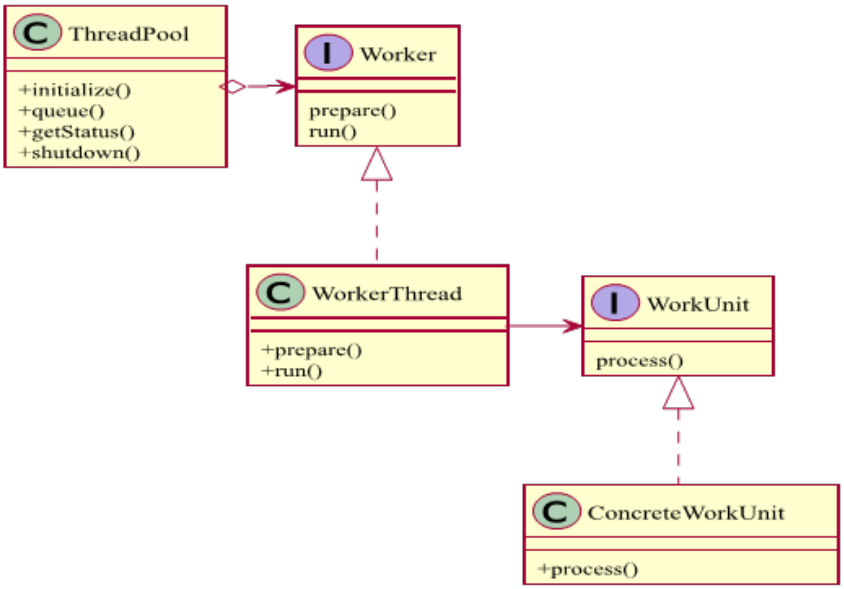


Figure 8.4: Thread pool pattern.

8.17 Remarks

- The thread pool pattern makes most sense with the task concept
- A task is a wrapper around a method call that could run concurrently
- In general we have more tasks than threads, i.e. tasks are executed by threads
- As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed
- The thread can then terminate, or sleep until there are new tasks available

8.18 Sample implementation

```
public interface IWorkUnit
{
    void Process();
}

public interface IWorker
{
    void Prepare();
    void Run();
    void Close();
}

public class WorkerThread : IWorker
```

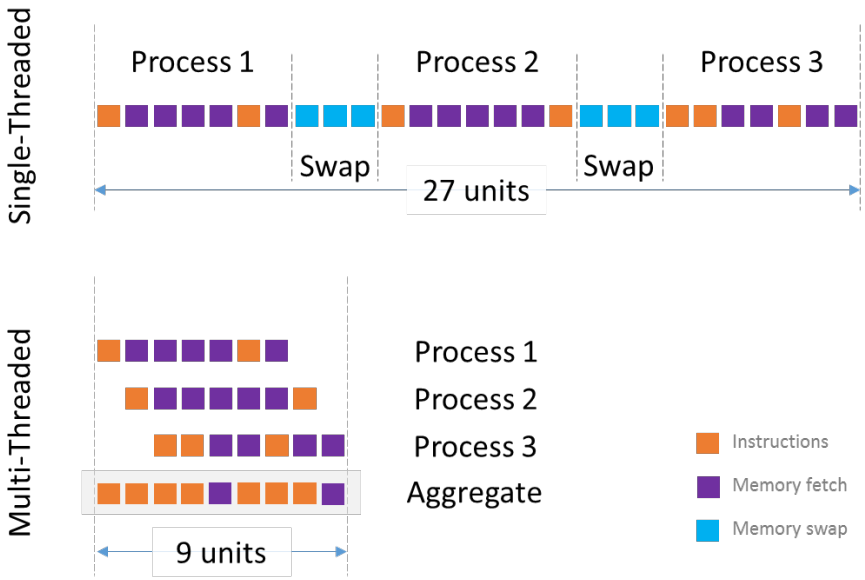
```
{
    Queue<IWorkUnit> queue;
    Thread thread;

    public WorkerThread(Queue<IWorkUnit> queue)
    {
        this.queue = queue;
    }

    public void Prepare()
    {
        thread = new Thread(Loop);
    }

    public void Run()
    {
        thread.Start();
    }

    public void Close()
    {
        thread.Abort();
    }

    void Loop()
    {
        while (true)
        {
            IWorkUnit item = null;

            lock(queue)
            {
                if (queue.Count > 0)
                    item = queue.Dequeue();
            }

            if (item != null)
                item.Process();
            else
                Thread.Sleep(100);
        }
    }
}

public class ThreadPool
{
    private readonly int nThreads;
    private readonly IWorker[] threads;
    private readonly Queue<IWorkUnit> queue;

    public ThreadPool(int nThreads)
    {
        this.nThreads = nThreads;
        this.queue = new Queue<IWorkUnit>();
        this.threads = new IWorker[nThreads];
    }

    public void Initialize()
```

```
    {
        for (int i = 0; i < nThreads; i++)
        {
            threads[i] = new WorkerThread(queue);
            threads[i].Prepare();
            threads[i].Run();
        }
    }

    public void Shutdown()
    {
        for (int i = 0; i < nThreads; i++)
        {
            threads[i].Close();
        }
    }

    public void Queue(IWorkUnit item)
    {
        lock(queue)
        {
            queue.Enqueue(item);
        }
    }
}
```

8.19 Practical considerations

- In .NET we could use the *ThreadPool* class
- However, for some problems our own thread pool might be useful
- Examples: Long running tasks, sleeping tasks with frequent polling
- The *WorkUnit* interface symbolizes a proper work item
- This allows our own thread pool to return results after completion
- Also we might implement dependencies from one *WorkUnit* to another
- This concept is implemented in .NET with the TPL

8.20 Advantages of multi-threading



8.21 The active object pattern

- The AO pattern decouples method execution from method invocation for an object
- The invocation should occur in the client's thread
- The execution in the AO thread of control
- The design should make this look transparent (using the proxy pattern)

8.22 Active object components

- **Proxy** Provides an interface to the client
- **Worker** base class with one class for each method of the proxy
- **Activation queue** that contains the requested invocations
- **Scheduler** that picks the request to be processed next
- **Servant** that processes the requests
- **Future** that contains the response

8.23 Active object diagram

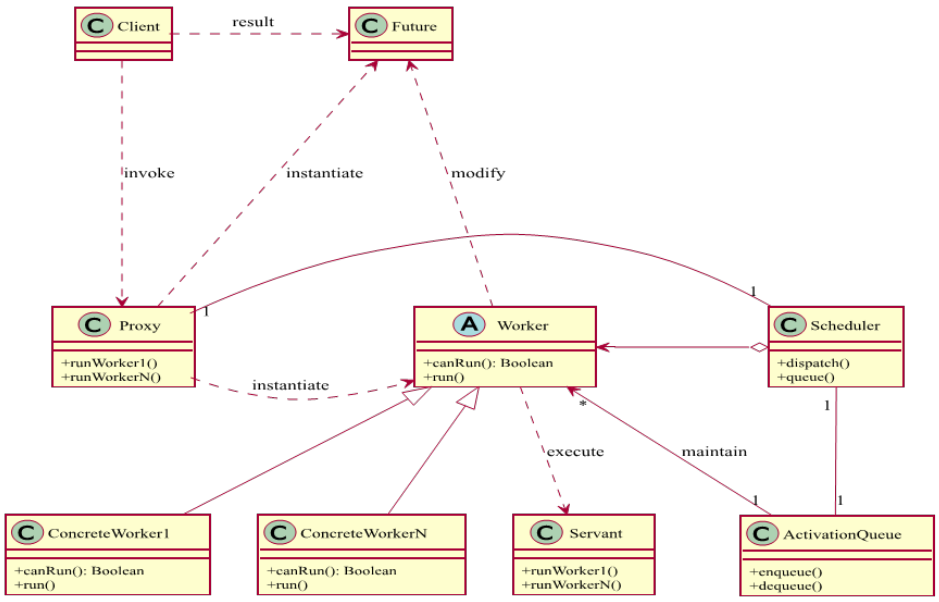


Figure 8.5: Active Object pattern.

8.24 Remarks

- The proxy transforms each request into an instance of a method class
- Possible parameters are stored / used for object creation
- It also creates the appropriate future for storing the result
- The future is initially empty and also has a status field
- The method object has a reference to the future
- The activation queue can be implemented using the Monitor pattern
- The servant has the same interface as the proxy and implements the actual methods

8.25 Sample implementation

```
class Future
{
  public bool IsFinished
  {
    get;
    private set;
  }

  public object Result
  {
```

```
    get;
    private set;
  }

  public void SetResult(object value)
  {
    if (IsFinished)
      return;

    IsFinished = true;
    Result = value;
  }
}

interface IWorker
{
  bool CanRun();

  void Run();

  Future Result { get; }
}

class Servant
{
  public double DoWork()
  {
    var sw = Stopwatch.StartNew();
    Console.WriteLine("I am now running ...");
    Thread.Sleep(1000);
    sw.Stop();
    return sw.ElapsedMilliseconds;
  }
}

class LongWorker : IWorker
{
  Servant servant;
  Future result;
  DateTime ahead;

  public LongWorker()
  {
    this.servant = new Servant();
    this.result = new Future();
    this.ahead = DateTime.Now.AddSeconds(10);
  }

  public Future Result
  {
    get { return result; }
  }

  public bool CanRun()
  {
    return DateTime.Now.CompareTo(ahead) >= 0;
  }
```

```
  public void Run()
  {
    var value = servant.DoWork();
    result.SetResult(value);
  }
}

class ShortWorker : IWorker
{
  Future result;

  public ShortWorker()
  {
    this.result = new Future();
  }

  public Future Result
  {
    get { return result; }
  }

  public bool CanRun()
  {
    return true;
  }

  public void Run()
  {
    result.SetResult("Short worker finished first!");
  }
}

class Scheduler
{
  List<IWorker> workers;
  static Scheduler current;
  Thread thread;
  object mlock;

  private Scheduler()
  {
    mlock = new object();
    workers = new List<IWorker>();
    thread = new Thread(Loop);
    thread.Start();
  }

  void Loop()
  {
    while (true)
    {
      Dispatch();
      Thread.Sleep(5);
    }
  }

  public static Scheduler Current
  {
```

```
    get { return current ?? (current = new Scheduler()); }
  }

  public void Stop()
  {
    thread.Interrupt();
    current = null;
  }

  public void Dispatch()
  {
    for (int i = 0; i < workers.Count; i++)
    {
      var worker = workers[i];

      if (worker.CanRun())
      {
        lock (mlock)
        {
          workers.RemoveAt(i);
        }

        worker.Run();
        Console.WriteLine("Worker finished!");
        break;
      }
    }
  }

  public void Queue(IWorker worker)
  {
    lock (mlock)
    {
      workers.Add(worker);
    }
  }
}

class Proxy
{
  public Future RunLongWorker()
  {
    var w = new LongWorker();
    Scheduler.Current.Queue(w);
    return w.Result;
  }

  public Future RunShortWorker()
  {
    var w = new ShortWorker();
    Scheduler.Current.Queue(w);
    return w.Result;
  }
}
```

8.26 Practical considerations

- The AO requires the implementation of many classes

- In particular, for every operation on the servant (and proxy) a worker class must be prepared
- Every such class must encode all the parameters of the operation and maintain a future
- Usually this is boring and repetitive code that could be generated
- This is therefore a good candidate for using templates
- Additionally we could take advantage of type lists

8.27 Sample activate object sequence

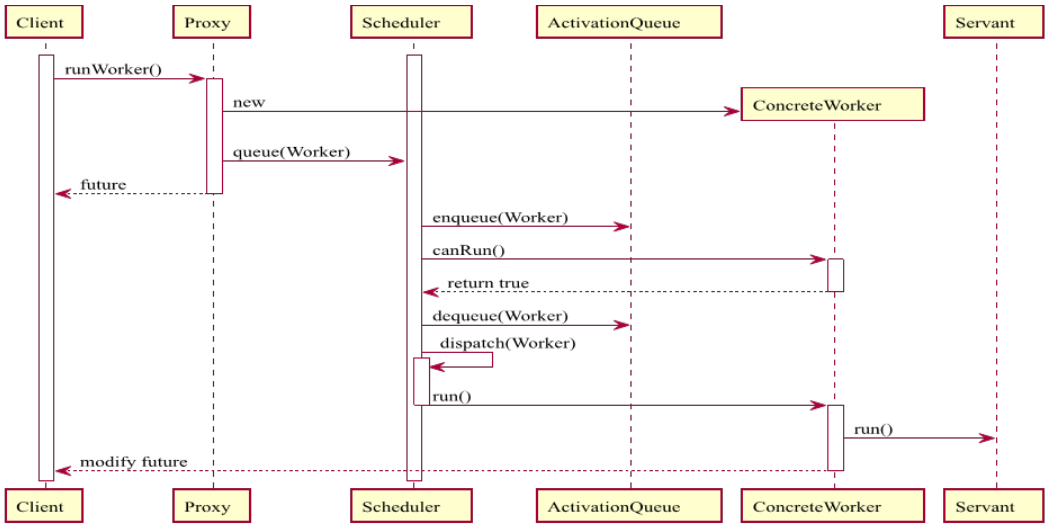


Figure 8.6: Active Object sample.

8.28 The reactor pattern

- GUI give us a single thread of control - the event loop or reactor
- The structure of such a reactor is as follows:
  1. **Resources** in form of *Handles*
  2. **Event Demultiplexer**, which uses the event loop to block all resources
  3. **Dispatcher**, registers / unregisters handlers and dispatches events
  4. **Event Handler**, with its assigned *Handle* (resource)
- The demultiplexer sends the handle to the dispatcher as soon as possible

8.29 Reactor diagram

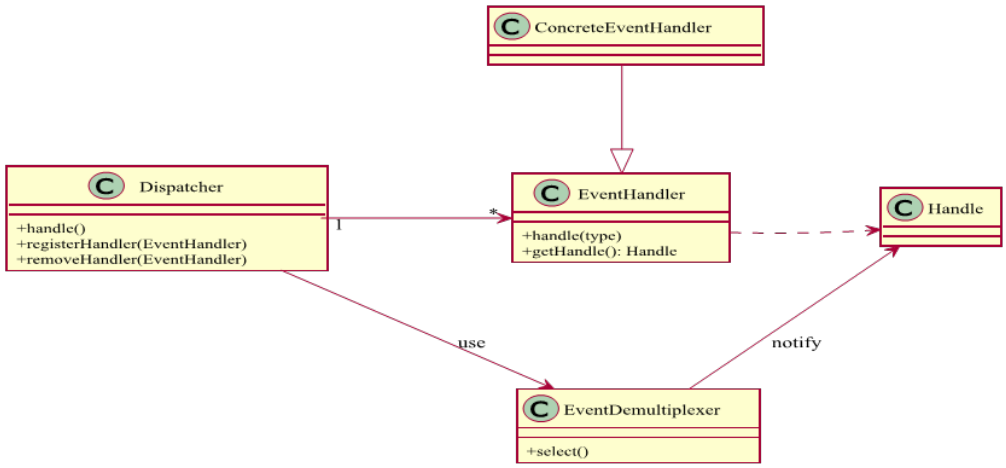


Figure 8.7: Reactor pattern.

8.30 Remarks

- A handle might be a system resource (connection, file, ...)
- All reactor systems are single-threaded
- However, they can exist in multi-threaded environments
- Usually communication with threads is possible by using channels
- JavaScript runs in a reactor, with code being single-threaded
- The reactor allows code to run concurrently without cross-threading

8.31 Sample implementation

```
class Handle
{
  //Empty for illustration
}

abstract class EventHandler
{
  public abstract Handle Handle { get; }

  public abstract void HandleEvent();
}

class ConcreteEventHandler : EventHandler
{
  private Handle myhandle;

  public override Handle Handle
```

```
  {
    get { return myhandle; }
  }

  public override void HandleEvent()
  {
    /* ... */
  }
}

class EventDemultiplexer
{
  object obj = new object();
  Queue<Handle> handles = new Queue<Handle>();

  public Handle Select()
  {
    lock (obj)
    {
      if (handles.Count > 0)
        return handles.Dequeue();
    }

    Thread.Sleep(100);
    return Select();
  }

  public void Notify(Handle myhandle)
  {
    lock (obj)
    {
      if (!handles.Contains(myhandle))
        handles.Enqueue(myhandle);
    }
  }
}

class Dispatcher
{
  List<EventHandler> handlers;

  public Dispatcher()
  {
    handlers = new List<EventHandler>();
  }

  public void RegisterHandler(EventHandler ev)
  {
    if (!handlers.Contains(ev))
      handlers.Add(ev);
  }

  public void RemoveHandler(EventHandler ev)
  {
    if (handlers.Contains(ev))
      handlers.Remove(ev);
  }
```

```
  public void Handle()
  {
    while (true)
    {
      var handle = source.Select();

      foreach (var handler in handlers)
        if (handler.Handle == handle)
          handler.HandleEvent();
    }
  }
}
```

8.32 Practical considerations

- Due to the synchronous calling of event handlers, the reactor gives us simple concurrency
- This is achieved without adding the complexity of multiple threads to the system
- However, the pattern itself is tedious to debug due to the inverted flow of control
- Additionally the single-threaded nature limits the maximum concurrency
- The scalability of the reactor pattern is also quite limited

8.33 Sample reactor sequence

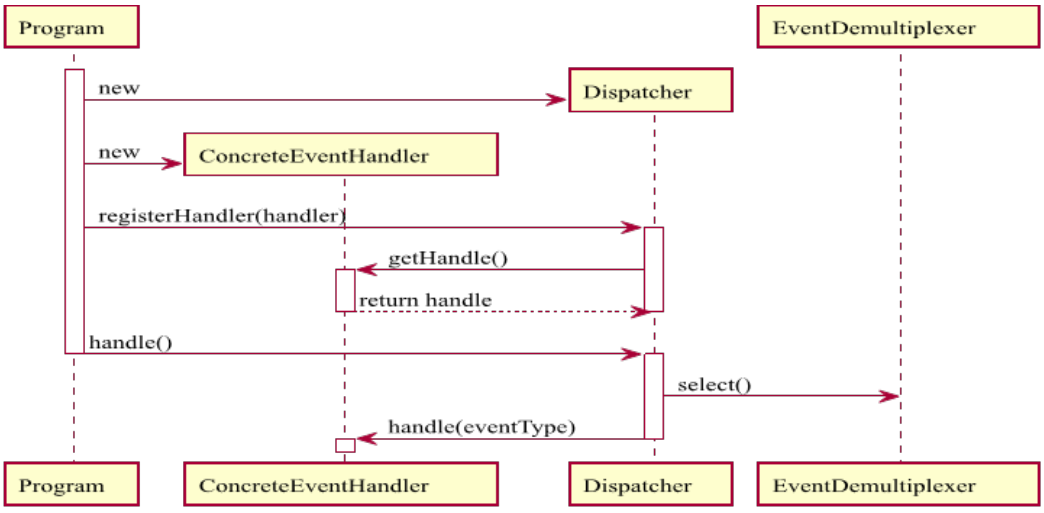


Figure 8.8: Reactor sample.

8.34 References

- Reactor Pattern Explained (`http://jeewanthad.blogspot.de/2013/02/reactor-pattern-explained-part-1.html`)

- The task pattern (http://www.developerdotstar.com/mag/articles/troche_taskpattern.html)
- MSDN: Concurrency Design Pattern (https://social.technet.microsoft.com/wiki/contents/articles/13210.concurrency-design-pattern.aspx)
- CodeProject: Windows Thread Pool (http://www.codeproject.com/Articles/6863/Windows-Thread-Pooling-and-Execution-Chaining)
- Wikipedia: Concurrency pattern (http://en.wikipedia.org/wiki/Concurrency_pattern)
- YouTube Playlist of Concurrency Design Patterns (http://www.youtube.com/playlist?list=PLmCsXDGbJHdhSdIWc9a4_ZXgqS0kvPX4t)
- What Every Dev Must Know About Multithreaded Apps (http://msdn.microsoft.com/en-us/magazine/cc163744.aspx)

8.35 Literature

- Lea, Doug (1999). *Concurrent Programming in Java: Design Principles and Patterns.*
- Schmidt, Douglas (1995). *Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events.*
- Schmidt, Douglas; Stal, Michael; Rohnert, Hans; Buschmann, Frank (2000). *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects.*

Chapter 9

Presentation patterns

9.1 Introduction

- Most applications consist of several layers:
  1. The **view** that is visible to the user
  2. The **data** that is presented or manipulated
  3. The **logic** that is responsible for driving the application
- Presentation patterns try to decouple these layers
- Additionally they aim for a maximum of flexibility
- Ideally they also support portability

9.2 Terminology

- The classes carrying the data are often called *models*
- A model is usually logic-free (a plain container)
- The various patterns now just differ in how these models are used
- Some use a *controller* to connect model and view
- Others use a *presenter* to actively modify the view with the model
- Then there are highly dynamic models that bridge the two like in MVVM

9.3 Model-View-Controller

- First appearance in the first GUI created by Xerox (Smalltalk)
- A **controller** connects model and view
- A **view** uses a model to generate an output representation
- A **model** contains information

- In a passive MVC the model is silent, however, in active MVC the model can send notifications (e.g. events) when changing
- These events can then be used by the controller or the view

9.4 MVC diagram

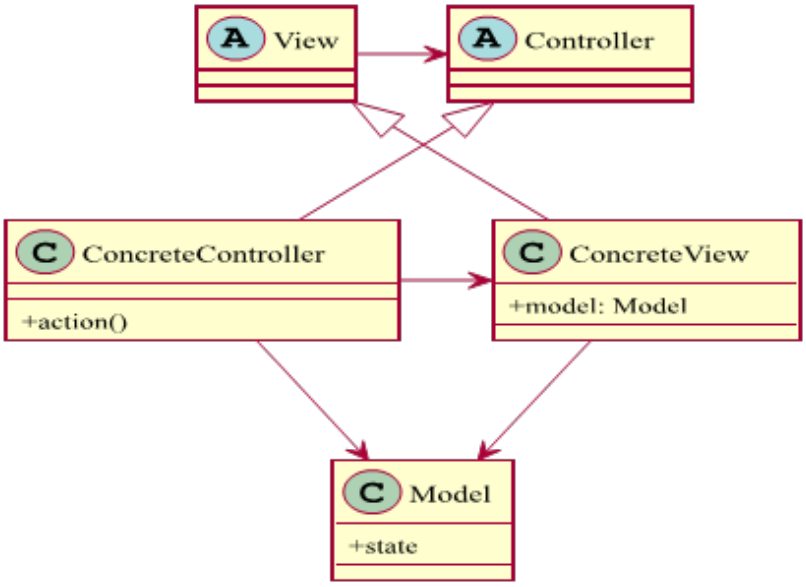
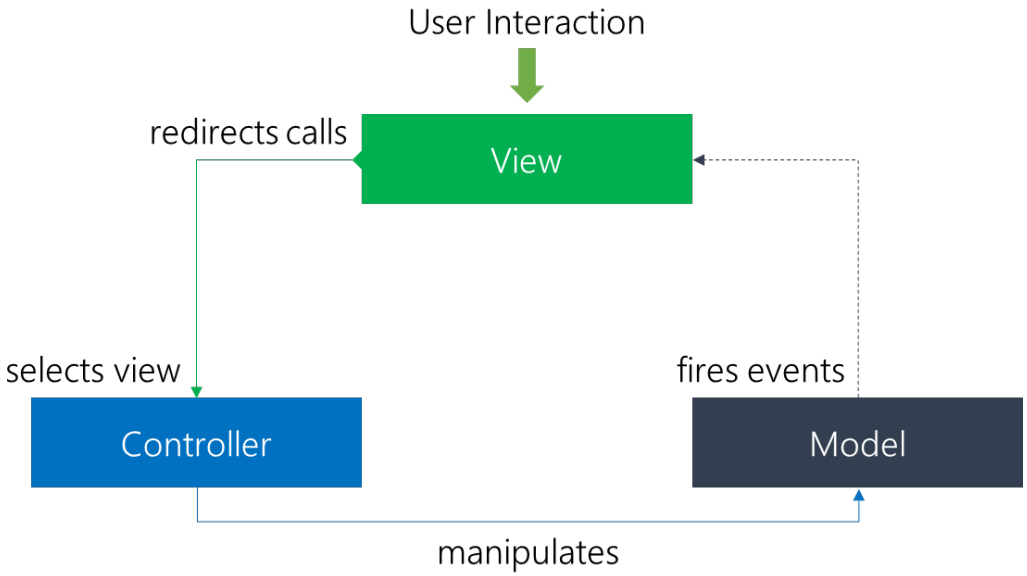


Figure 9.1: MVC.

9.5 Remarks

- Different aspects of an application are kept separate by using MVC
- Benefits of such a separation:
  1. The separation of concerns is required for decoupling
  2. An application might have more than one representation (client, console, web) and the elements of the user interface need to be kept separate from the parts that are common to each
  3. Different developers (with different skills) may be responsible for different aspects of the application (e.g. designers for the view)

9.6 MVC in action



9.7 Model and view

- Model:
  - Represents data and rules that govern access to and updates of it
  - Simple real-world modeling techniques apply when defining the model, since usually it is basically a real-world approximation
- View:
  - Renders the contents of a model
  - Accesses data through the model and chooses data representation
  - Responsible for maintaining consistency when the model changes

9.8 The controller

- The controller translates interactions with the view into actions
- These actions are performed by the model
- e.g. in a stand-alone GUI client, user interactions could be button clicks, whereas in a web app, they appear as HTTP requests
- The actions performed by the model include activating business processes or changing the state of the model
- Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view

9.9 Web implementation

- A class (called router) is required to interpret incoming requests and direct them to the appropriate controller
- The corresponding method (called action) is then called
- The controller might update the model based on the request
- Finally a response in chosen in form of a view, which might be HTML
- This requires access to some class that can be used to display the appropriate view

9.10 Guess a number

```
public interface IView<T> where T : Model
{
  T Model { get; set; }

  BaseController Controller { get; set; }
}

public abstract class BaseController
{
  public abstract void Command(string value);
}

public class Model
{
  public event EventHandler Changed;

  protected void RaiseChanged()
  {
    if (Changed != null)
      Changed(this, EventArgs.Empty);
  }
}

class GuessModel : Model
{
  string name;
  int maxtrials;
  int numtrials;
  string lastname;

  public string Name
  {
    get { return name; }
    set { name = value; RaiseChanged(); }
  }

  public bool IsCorrect
  {
    get { return lastname == name; }
  }
```

```
  public int MaxTrials
  {
    get { return maxtrials; }
    set { maxtrials = value; RaiseChanged(); }
  }

  public int NumTrials
  {
    get { return numtrials; }
  }

  public void EnterTrial(string name)
  {
    numtrials++;
    lastname = name;
    RaiseChanged();
  }
}

public abstract class ConsoleView<T> : IView<T> where T : Model
{
  T model;

  public T Model
  {
    get { return model; }
    set
    {
      if (model != null) model.Changed -= ModelChanged;
      if (value != null) value.Changed += ModelChanged;

      model = value;
      RaiseModelChanged(model);
    }
  }

  public BaseController Controller { get; set; }

  protected void PrintOutput(string output)
  {
    Console.WriteLine(output);
  }

  void ModelChanged(object sender, EventArgs e)
  {
    RaiseModelChanged(model);
  }

  protected abstract void RaiseModelChanged(T model);
}

class GuessMyNameView : ConsoleView<GuessModel>
{
  protected override void RaiseModelChanged(GuessModel model)
  {
    if (!model.IsCorrect)
    {
```

```
      if (model.NumTrials == model.MaxTrials)
      {
        PrintOutput(string.Format("The name {0} would have been correct!", model
            .Name));
      }
      else
      {
        PrintOutput(string.Format("Trial {0} / {1}. What's the name?",
          model.NumTrials + 1, model.MaxTrials));
        var input = Console.ReadLine();
        Controller.Command(input);
      }
    }
    else
    {
      PrintOutput(string.Format("The name {0} is correct (guessed with {1}
          trials)!",
          model.Name, model.NumTrials));
    }
  }
}

class GuessMyNameController : BaseController
{
  IView<GuessModel> view;
  GuessModel model;

  public GuessMyNameController(IView<GuessModel> view, GuessModel model)
  {
    this.view = view;
    this.model = model;
    this.view.Controller = this;
  }

  public void Start()
  {
    view.Model = model;
  }

  public override void Command(string value)
  {
    model.EnterTrial(value);
  }
}
```

9.11 Practical considerations

- Today MVC is mostly used in web development
- Popular frameworks contain a very useful basis
  - ASP.NET MVC (.NET)
  - Rails (Ruby)
  - Spring (Java)
  - AngularJS (JavaScript)

  - CakePHP (PHP)
- Usually writing an MVC framework from scratch is not recommended

9.12 Model-View-Presenter

- In contrast to MVC, all presentation logic is pushed to the presenter
- A **model** is an interface defining the data to be displayed
- A **view** is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data
- A **presenter** acts upon the model and the view
- The presenter formats data for display in the view
- There is a single presenter for each view

9.13 MVP diagram

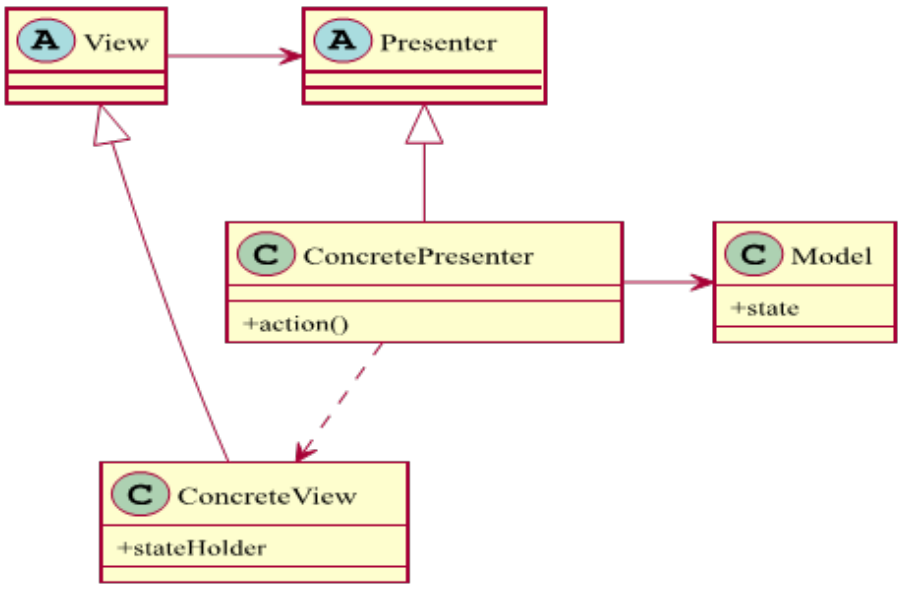
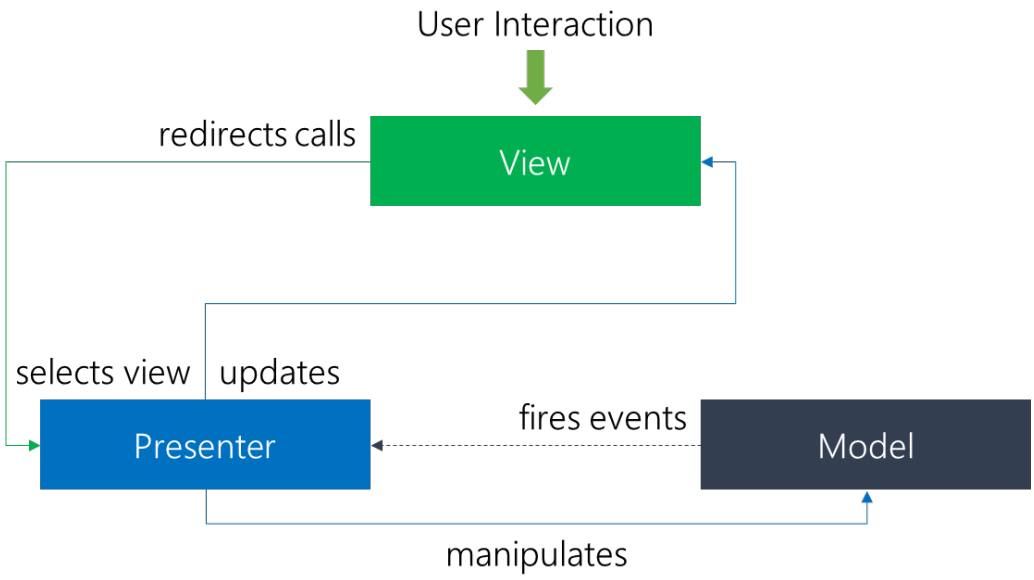


Figure 9.2: MVP.

9.14 Remarks

- Real two-way communication with the view
- Every view implements some sort of View interface
- In the view an instance of the presenter is referenced
- Events are forwarded to the presenter by the view
- The view never passes view related code (e.g. UI controls) to the presenter

9.15 MVP in action



9.16 Model and view

- Model:
  - Communication with DB layer
  - Raising appropriate event when dataset is generated
- View:
  - Renders data
  - Receives events and represents data
  - Have basic validations (e.g. invalid email, ...)

9.17 The presenter

- Decouples a concrete view from the model
- Supports view in complex decisions
- Communicate with model
- Complex validations (e.g. involve other data sources)
- Queries model
- Retrieves data from model, formats it and sends them to the view
- The view is updated through the same event approach that MVC uses

9.18 Implementation concept

```
public interface IUserView
{
    void ShowUser(User user);
}

public partial class UserForm : Form, IUserView
{
    UserPresenter _presenter;

    public UserForm()
    {
        _presenter = new UserPresenter(this);
        InitializeComponent();
    }

    private void SaveUser_Click(object sender, EventArgs e)
    {
        User user = ConstructUser();
        _presenter.SaveUser(user);
    }

    /* ... */
}

public class UserPresenter
{
    IUserView _view;

    public UserPresenter(IUserView view){
        _view = view;
    }

    public void SaveUser(User user)
    {
  /* ... */
    }

    /* ... */
}
```

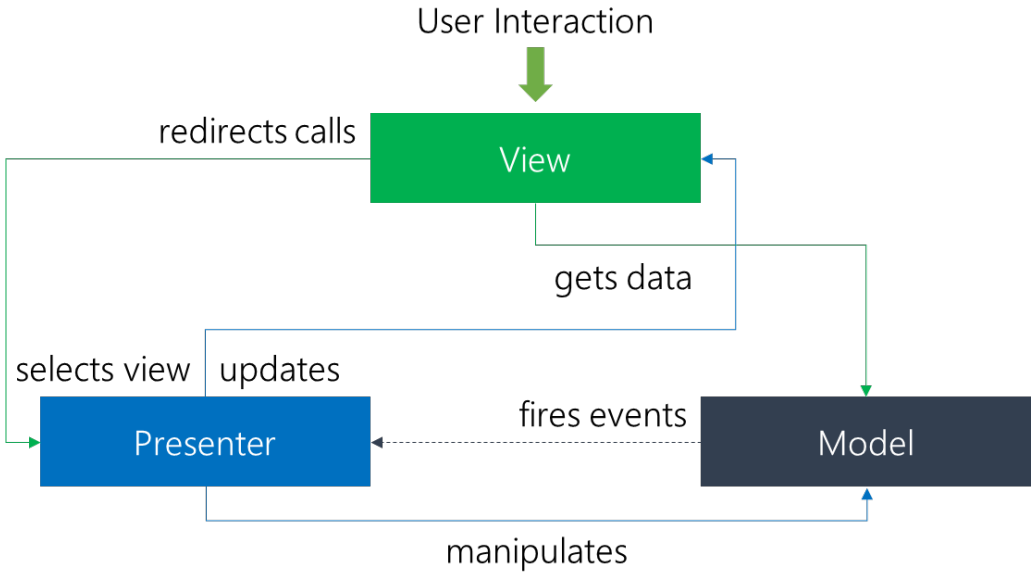
9.19 Practical considerations

- Today MVP is mostly used in client development
- Popular frameworks contain a very useful basis like MVC# (.NET, strange name) or GWT (Java for web development)
- In general the difference to MVC is subtle
- Sometimes the general outline looks like MVP, but it is in fact MVC
- As with MVC it is not necessary to start writing a custom framework
- Today MVVM is more popular for client development

9.20 MVP modes

- There are two main modes for MVP:
  1. Passive view
     - Interaction is only handled by the presenter
     - View is updated exclusively by the presenter
  2. Supervising controller
     - The view interacts with the model (simple binding)
     - View is updated by the presenter through data-binding

9.21 Supervising MVP



9.22 Model-View-ViewModel

- MVVM tries to gain the advantages of MVC (separation) and data-binding
- A **model** has practically the same role as a model in MVC
- The **view** is also just the pure representation as with MVC
- A **ViewModel** is an abstraction of the view, that serves as target for data binding (kind of a controller, but only for data conversion)
- The role of the controller or presenter is now outsourced in a generic **Binder**, which is responsible for updating the UI or model (two-way)
- The binder is provided by the framework and should not be touched

9.23 MVVM diagram

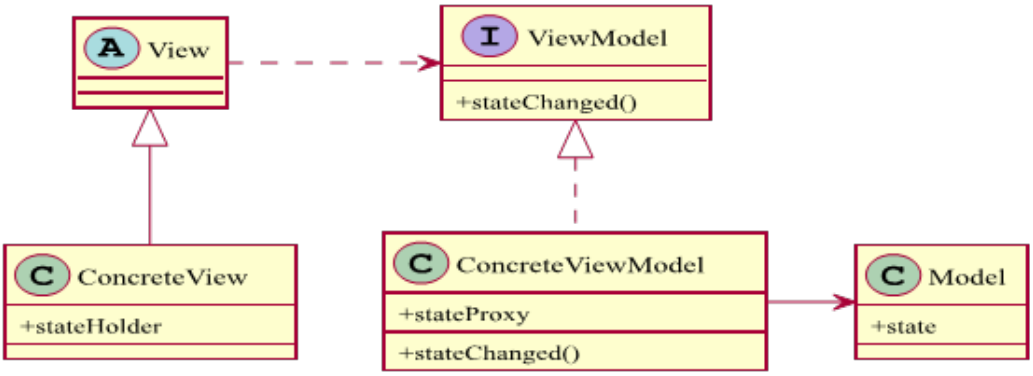
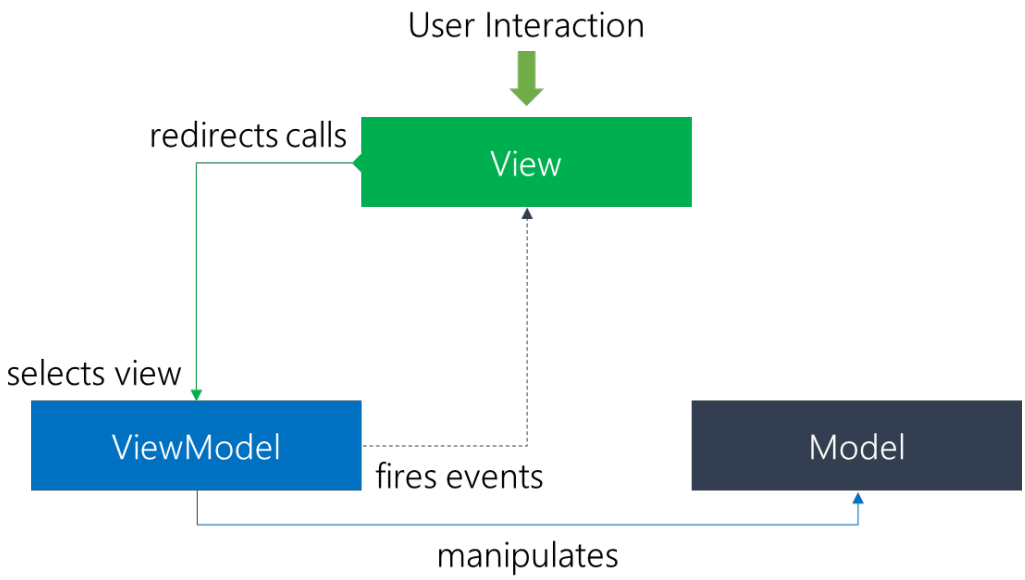


Figure 9.3: MVVM.

9.24 Remarks

- First implementation of MVVM is in the WPF framework
- Here we have two-way communication with the view
- The VM represents the view in a representation independent way
- The view binds directly to the ViewModel
- A single ViewModel should be used for each view
- Since the binder is the actual key part the pattern is sometimes called **MVB** (Model-View-Binder)

9.25 MVVM in action



9.26 The ViewModel

- It is the model of the view
- Could be seen as a mediator between view and model
- Contains the generic data binding capabilities that can be used by the view (binding to the conceptual state of the data)
- It could be seen as a specialized aspect of what would be a controller in the original MVC pattern
- Acts as a converter that changes model information into view information and passes commands from the view into the model

9.27 A very simple ViewModel

```
public class StudentData : INotifyPropertyChanged
{
    string _firstName;
    double _gradePointAverage;

    public event PropertyChangedEventHandler PropertyChanged;

    void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }

    public string StudentFirstName
    {
        get
        {
            return _firstName;
        }
        set
        {
            _firstName = value;
            OnPropertyChanged("StudentFirstName");
        }
    }

    public double StudentGradePointAverage
    {
        get
        {
            return _gradePointAverage;
        }

        set
        {
            _gradePointAverage = value;
            OnPropertyChanged("StudentGradePointAverage");
        }
```

```
    }
}
```

9.28 Practical considerations

- Usually one does not directly implement the interface
- Abstract base classes with useful helpers are preferred
- Instead of naming the changed property with a string strong names should be used (compiler generated or reflection)
- Commands also need some kind of special interface, which is usually called *ICommand* or similar
- A command usually consists of a run and canrun method
- The latter determines the state of buttons which invoke the command

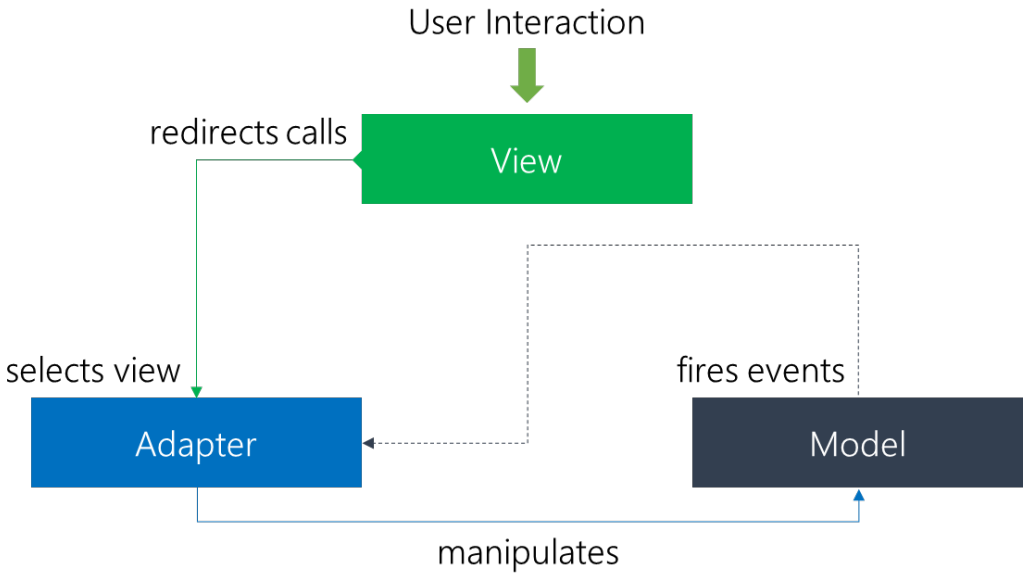
9.29 When to use what

- **MVC**
  - Connection between the displayed view and the rest is impossible
  - If there could be no binding context
- **MVP**
  - General binding is not possible, but specialized binding is
  - Use in situations like MVC, but where a connected view exists
- **MVVM**
  - General binding is possible and realized automatically
  - The view is directly connected and highly interactive

9.30 More patterns

- There are many more presentation patterns
- However, differences are even more subtle
- The two most important other presentation patterns:
  1. the MVA (Model-View-Adapter) and
  2. the PCA (Presentation-Abstraction-Control)
- They are also quite close to the original MVC pattern
- It is more natural to see them as an extension of MVC

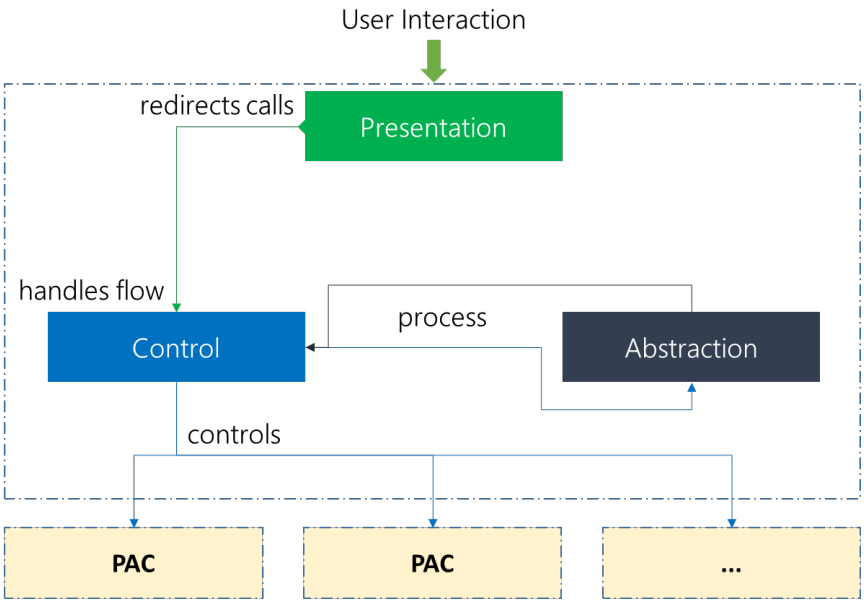
9.31 Model-View-Adapter



9.32 Remarks

- Here the adapter is a real adapter, acting as a mediator between model and view
- This breaks the classic MVC triangle
- Chance: The view is completely decoupled from the model
- Pitfall: The adapter is strongly coupled on both
- To solve this one might use multiple lightweight adapters

9.33 PresentationAbstractionControl



9.34 Remarks

- Quite close to MVA, however, control and abstraction are stronger coupled
- Main difference: Division into independent PCA cells
- The abstraction component retrieves and processes the data
- Generally this is an hierarchical layout
- A PCA controller can with other PCA controllers (children)
- Addtionally it handles the flow of control and communication between presentation and abstraction

9.35 References

- MVVM vs MVP vs MVC (`http://joel.inpointform.net/software-development/mvvm-vs-mvp-vs-mvc-the-differences-explained/`)
- The MVA pattern (`http://www.palantir.com/2009/04/model-view-adapter/`)
- Presentation-Abstraction-Control in Java (`http://www.vico.org/pages/PatronsDisseny/Pattern%20Presentation%20Abstra/`)
- CodeProject: MVC with Windows Forms (`http://www.codeproject.com/Articles/383153/The-Model-View-Controller-MVC-Pattern-with-Csharp`)
- Wikipedia: MVC (`http://en.wikipedia.org/wiki/ModelâĂŞviewâĂŞcontroller`)
- Channel9: Understanding the MVVM Pattern (`http://channel9.msdn.com/Events/MIX/MIX10/EX14`)

9.36 Literature

- Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael (1996). *Pattern-Oriented Software Architecture Vol 1: A System of Patterns.*

- Garofalo, Raffaele (2011). *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern.*

- Guermeur, Daniel; Unruh, Amy (2010). *Google App Engine Java and GWT Application Development.*

Chapter 10

SOLID principles

10.1 Introduction

- SOLID is a mnemonic acronym introduced in the early 2000s
- Basically this contains the five basic principles of object-oriented programming and design
- These principles are meant to guide programmers into better designs
- Mastering SOLID is considered one of the most important abilities
- Applying the techniques presented in this chapter should result in better programs

10.2 The five categories

- **SOLID** is composed of
  - **S**RP (*Single Responsibility Principle*)
  - **O**CP (*Open Closed Principle*)
  - **L**SP (*Liskov Substitution Principle*)
  - **I**SP (*Interface Segregation Principle*)
  - **D**IP (*Dependency Inversion Principle*)
- To put it into one sentence: Classes should be decoupled, reusable, closed yet extendable and only responsible for a single task

10.3 Single responsibility principle

- This principle can be boiled down to the following:

  ” A class should have only a single responsibility.“
- Why is that important? Reducing complexity! This reduces errors
- Also this makes the class more pluggable

- And finally the implementation is probably more dedicated and lightweight (plus easier to test)

10.4 Open / closed principle

” Software entities... should be open for extension, but closed for modification. “

- This means that classes should be open for extension e.g. in form of inheritance
- However, the core purpose of the class should be closed for modification
- We are not able to modify the class in such a way, that the original purpose is no longer fulfilled
- We have seen patterns, e.g. Strategy pattern, which embrace this

10.5 Liskov substitution principle

- Making the closed part of OCP even stronger:

  ” Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.“
- This is also stated in the design by contract philosophy
- Basically only new methods should be introduced by subtyping
- By replacing a method one should not change the core behavior

10.6 Interface segregation principle

” Many client-specific interfaces are better than one general-purpose interface. “

- Basically we are interested in composing the right interfaces
- We are not interested in leaving many methods unimplemented
- As with SRP, interfaces should be as atomic as possible
- Intention to create a barrier for preventing dependencies

10.7 Dependency inversion principle

” Depend upon Abstractions. Do not depend upon concretions. “

- This is actually one of the most crucial points in OOP
- It inverts the way someone may think about OO design by dictating that both high- and low-level objects must depend on the same abstraction

- Depending on concrete implementations yields strong coupling
- Changing the implementation might cause bugs in the dependencies
- Dependency injection is a method build upon this principle

10.8 Abstraction

- Obviously abstraction is important - most patterns introduced some
- It helps to identify the important aspects and ignore the details
- Supports separation of concerns by divide & conquer vertically
- Abstractions dominate computing
  - Design Models (ER, UML etc)
  - Programming Languages (C#, C++, Java, etc.)
- This is *control abstraction*, but there is also *data abstraction*

10.9 Why the separation?

- We cannot deal with all aspects of a problem simultaneously
- To conquer complexity, we need to separate issues and tasks
  - Separate functionality from efficiency
  - Separate requirements specification from design
  - Separate responsibilities
- Today's applications involve interoperability of
  - Client/Server, Legacy system, COTS (3rd party), databases, etc.
  - Multiple programming languages (C#, C++, Java, etc.)
  - Heterogeneous hardware/OS platforms

10.10 Extending classes

- The OCP is a very important principle
- A class should be open for extension, but closed for modification
- In other words, (in an ideal world...) we should never need to change existing code or classes
- Exception: Bug-fixing and maintainance
- All new functionality can be added by adding new subclasses and overriding methods
- Alternatively by reusing existing code through delegation

10.11 SRP example

(Image courtesy of Derick Bailey.)

10.12 Where SRP should be applied

```
public class ServiceStation
{
    public void OpenGate()
    {
        /* ... */
    }

    public void DoService(Vehicle vehicle)
    {
        /* ... */
    }

    public void CloseGate()
    {
        /* ... */
    }
}
```

10.13 Why SRP makes sense here

- The class has two responsibilites:
    1. Performing the service
    2. Controlling the gate
- However, the name suggests only one responsibility

- Maintenance is much easier if we just deal with a set of methods that belong to the same group
- Hence we should outsource the methods for controlling the gate
- In order to keep the coupling at a minimum we should use an interface

10.14 With SRP in action

```
public interface IGateUtility
{
    void OpenGate();
    void CloseGate();
}

public class ServiceStationUtility : IGateUtility
{
    public void OpenGate()
    {
        /* ... */
    }

    public void CloseGate()
    {
        /* ... */
    }
}

public class ServiceStation
{
    IGateUtility _gateUtility;

    public ServiceStation(IGateUtility gateUtility)
    {
        this._gateUtility = gateUtility;
    }
    public void OpenForService()
    {
        _gateUtility.OpenGate();
    }

    public void DoService()
    {
        /* ... */
    }

    public void CloseForDay()
    {
        _gateUtility.CloseGate();
    }
}
```

10.15 OCP example

(Image courtesy of Derick Bailey.)

10.16 Where OCP should be applied

```
public class MileageCalculator
{
    IEnumerable<Car> _cars;

    public MileageCalculator(IEnumerable<Car> cars)
    {
        this._cars = cars;
    }

    public void CalculateMileage()
    {
        foreach (var car in _cars)
        {
            if (car.Name == "Audi")
                Console.WriteLine("Mileage of the car {0} is {1}", car.Name, "10
                    M");
            else if (car.Name == "Mercedes")
                Console.WriteLine("Mileage of the car {0} is {1}", car.Name, "20
                    M");
        }
    }
}
```

10.17 Why OCP makes sense here

- The class represents a typical mess for maintenance and extension
- For two names the method seems OK, however, consider many more

- Additionally every new name will result in changing the method
- While the closed principle is violated, the open principle is also violated
- The mileage is provided as a hardcoded string
- There is also no possibility to give special cars special mileages

10.18 With OCP in action

```
public interface ICar
{
    string Name { get; set; }
    string GetMileage();
}

public class Audi : ICar
{
    public string Name { get; set; }

    public string GetMileage()
    {
        return "10M";
    }
}

public class Mercedes : ICar
{
    public string Name { get; set; }

    public string GetMileage()
    {
        return "20M";
    }
}

public class CarController
{
    List<ICar> cars;

    public CarController()
    {
        cars = new List<ICar>();
        cars.Add(new Audi());
        cars.Add(new Mercedes());
    }

    public string GetCarMileage(string name)
    {
        if (!cars.Any(car => car.Name == name))
            return null;

        return cars.First(car => car.Name == name).GetMileage();
    }
}
```

```
public class MileageCalculator
{
    IEnumerable<Car> _cars;
    CarController _ctrl;

    public MileageCalculator(IEnumerable<Car> cars)
    {
        this._cars = cars;
        this._ctrl = new CarController();
    }

    public void CalculateMileage()
    {
        foreach (var car in _cars)
        {
            var mileage = _ctrl.GetCarMileage(car.Name);

            if (mileage != null)
                Console.WriteLine("Mileage of the car {0} is {1}", car.Name,
                    mileage);
        }
    }
}
```

10.19 LSP example

(Image courtesy of Derick Bailey.)

10.20 Where LSP should be applied

```
public class Apple
{
    public virtual string GetColor()
    {
```

```
        return "Red";
    }
}

public class Orange : Apple
{
    public override string GetColor()
    {
        return "Orange";
    }
}

//Possible usage:
Apple apple = new Orange();
Console.WriteLine(apple.GetColor());
```

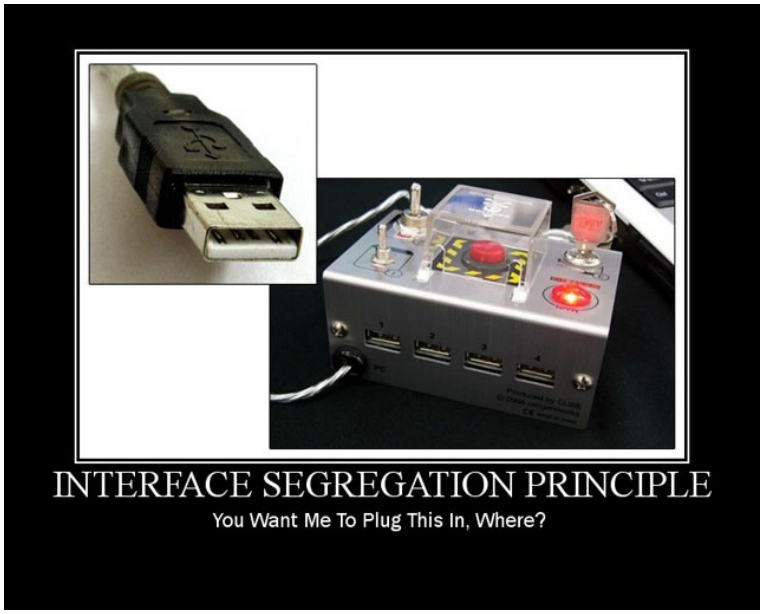
10.21 Why LSP makes sense here

- This case appears quite often: One derives from a somehow related, but conceptually different class
- Problem? A method that accepts an apple could then receive an orange (but some methods may not like oranges and will be surprised by the taste of this "apple" ...)
- Here we cannot say an orange is an apple, but both are definitely fruits
- A common base class (which will be abstract in most cases) is missing
- Always think about related and unrelated, parallel and derived

10.22 With LSP in action

```
public abstract class Fruit
{
    public abstract string GetColor();
}

public class Apple : Fruit
{
    public override string GetColor()
    {
        return "Red";
    }
}

public class Orange : Fruit
{
    public override string GetColor()
    {
        return "Orange";
    }
}
```

10.23 ISP example



(Image courtesy of Derick Bailey.)

10.24 Where ISP should be applied

```
public interface IOrder
{
    void Purchase();
    void ProcessCreditCard();
}

public class OnlineOrder : IOrder
{
    public void Purchase()
    {
        //Do purchase
    }

    public void ProcessCreditCard()
    {
        //process through credit card
    }
}

public class InpersonOrder : IOrder
{
    public void Purchase()
    {
        //Do purchase
    }

    public void ProcessCreditCard()
    {
        //Not required for inperson purchase
```

```
        throw new NotImplementedException();
    }
}
```

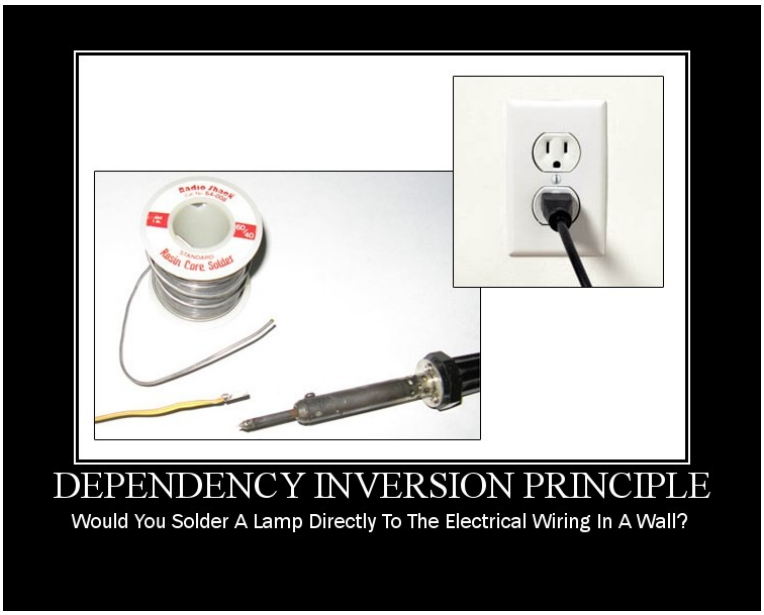
10.25 Why ISP makes sense here

- Sometimes not-implementing a full interface makes sense
- However, if we define the interfaces we should define them in such a way that only contains the bare methods
- In this example the interface is definitely too broad, we need to separate it into two two interfaces
- Sometimes it also might make sense to mark the second interface as being dependent on the first one (staging the interface by implementing other interfaces)

10.26 With ISP in action

```
public interface IOrder
{
    void Purchase();
}

public interface IOnlineOrder
{
    void ProcessCreditCard();
}

public class OnlineOrder : IOrder, IOnlineOrder
{
    public void Purchase()
    {
        //Do purchase
    }

    public void ProcessCreditCard()
    {
        //process through credit card
    }
}

public class InpersonOrder : IOrder
{
    public void Purchase()
    {
        //Do purchase
    }
}
```

10.27 DIP example



(Image courtesy of Derick Bailey.)

10.28 Where DIP should be applied

```
class EventLogWriter
{
    public void Write(string message)
    {
        //Write to event log here
    }
}

class AppPoolWatcher
{
    EventLogWriter writer = null;

    public void Notify(string message)
    {
        if (writer == null)
        {
            writer = new EventLogWriter();
        }
        writer.Write(message);
    }
}
```

10.29 Why DIP makes sense here

- The class looks quite fine already, however, it is violating DIP
- The high level module AppPoolWatcher depends on EventLogWriter which is a concrete class and not an abstraction

- Future requirements might break the code on this position (what about other kinds of loggers?)
- The solution for fixing this problem is called *inversion of control*

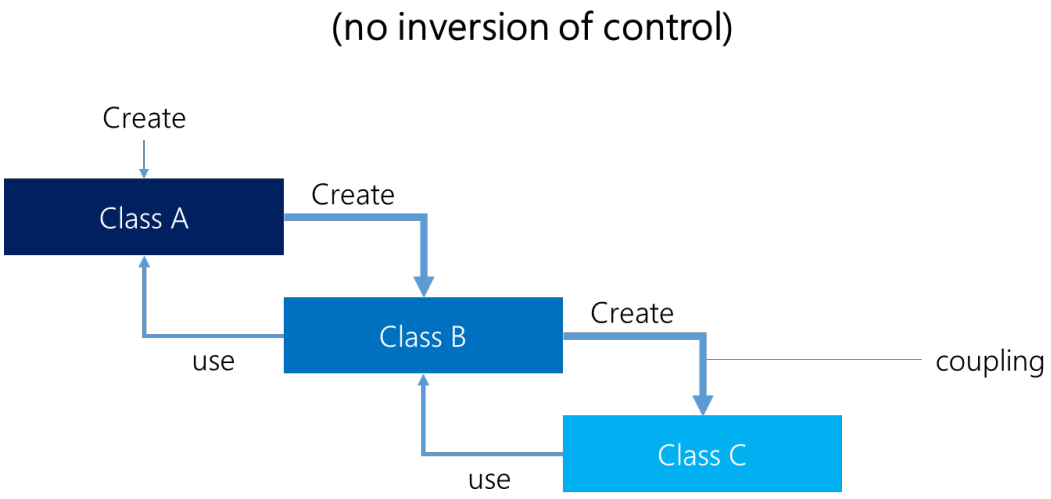
10.30 With DIP in action

```
public interface INofificationAction
{
    public void ActOnNotification(string message);
}

class EventLogWriter : INofificationAction
{
    public void ActOnNotification(string message)
    {
        /* ... */
    }
}

class AppPoolWatcher
{
    INofificationAction action = null;

    public AppPoolWatcher(INofificationAction concreteImplementation)
    {
        this.action = concreteImplementation;
    }

    public void Notify(string message)
    {
        action.ActOnNotification(message);
    }
}
```

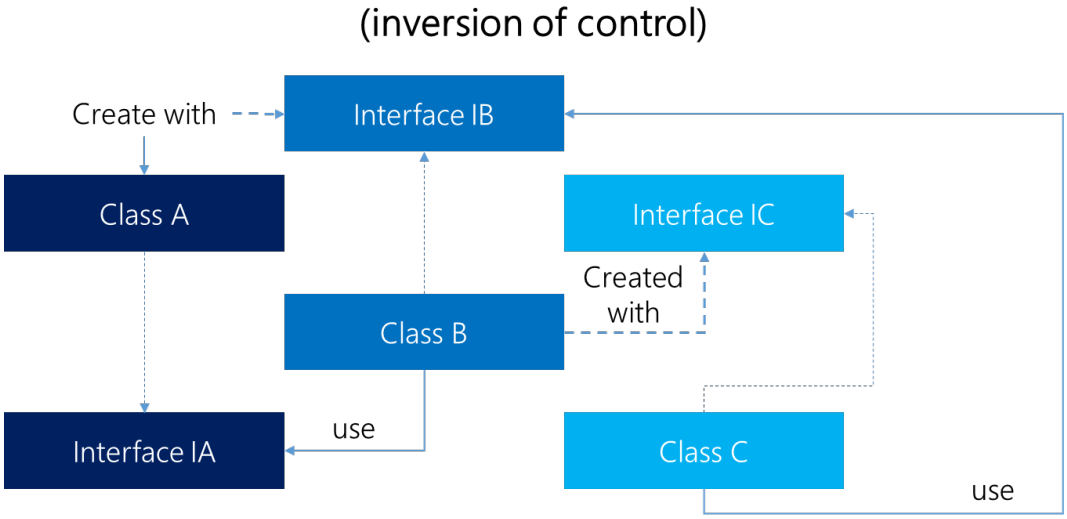
10.31 Dependency Injection

- In the last example we prepared everything for dependency injection, which might appear as
  1. Constructor injection (see last example)
  2. Method injection
  3. Property injection
- DI is mainly for injecting the concrete implementation into a class that is using abstraction i.e. interface inside
- Moving out the concrete implementation and providing a way for externally providing concrete implementations is the idea of DI

10.32 Strong coupling



10.33 Inversion of control



10.34 References

- CodeProject: SOLID architectures using simple C# (`http://www.codeproject.com/Articles/703634/SOLID-Architecture-principles-using-simple-Csharp`)
- CodeProject: An Absolute Beginner's Tutorial on DIP (`http://www.codeproject.com/Articles/615139/An-Absolute-Beginners-Tutorial-on-Dependency-Inver`)
- SOLID Development Principles In Motivational Pictures (`http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-p`
- CodeGuru: SOLID principles overview (`http://www.codeguru.com/columns/experts/solid-principles-in-c-an-overview.htm`)

- Wikipedia: SOLID (`http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29`)
- Uncle Bob: Principles Of OOD (`http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`)

10.35 Literature

- Riel, Arthur J (1996). *Object-Oriented Design Heuristics.*
- Plauger, P.J. (1993). *Programming on Purpose: Essays on Software Design.*
- Larman, Craig (2001). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.*
- Martin, Robert C. (2002). *Agile Software Development, Principles, Patterns, and Practices.*
- DeMarco, Tom. (1979). *Structured Analysis and System Specification.*

Chapter 11

Best practices

11.1 Introduction

- In this section best practices will be shown for:
  - object-oriented design,
  - code transformations (refactoring),
  - robustness (test driven development),
  - performance (device independent) and
  - readability.
- The content is complementary to the previous chapters

11.2 Levels of design

- Creating software is much more than just programming
- A lot of time needs to be taken for designing an application
- There are various levels of architecture:
  - System
  - Packages (e.g. business rules, UI, DB, dependencies on the system)
  - Classes
  - Routines
  - Logic / algorithm

11.3 Desirable characteristics of a design

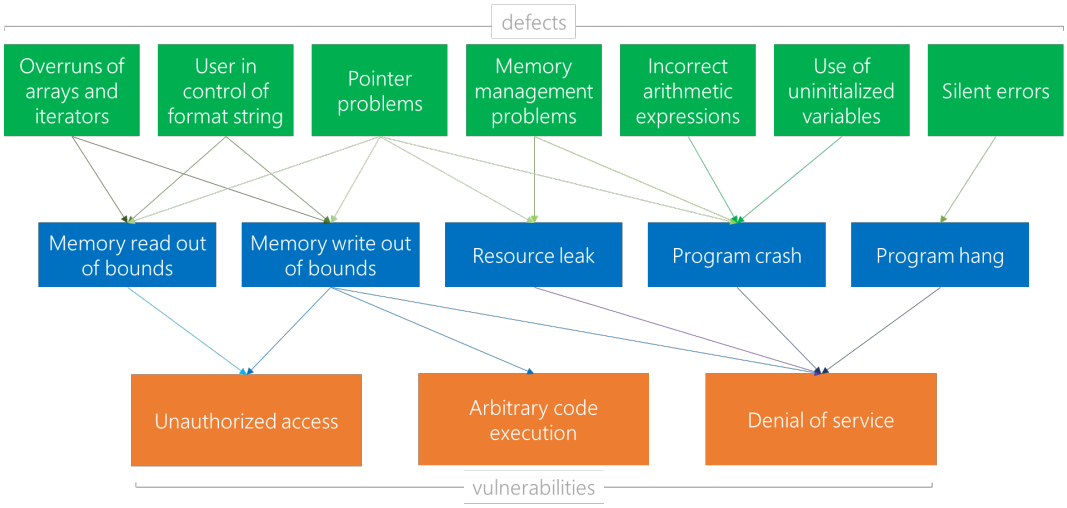
- Minimal complexity
- Ease of maintenance
- Loose coupling (good abstraction, information hiding)

- Extensibility and reusability
- High fan-in (large number of classes that use a given class)
- Low fan-out (a given class should not use too many other classes)
- Portability
- Leanness, i.e. no extra parts, backward-compatible
- Stratification: The system must be consistent at any level

11.4 Defensive programming

- Protecting from invalid input
- Unit testing
- Error-handling (e.g. return neutral value, error code, same answer as last time, ...)
- Robustness and correctness
- Exceptions are introduced if wrong inputs happen
- Being not too defensive is key (remove trivial error checking from production code, ...)

11.5 Common errors



11.6 Test-driven development

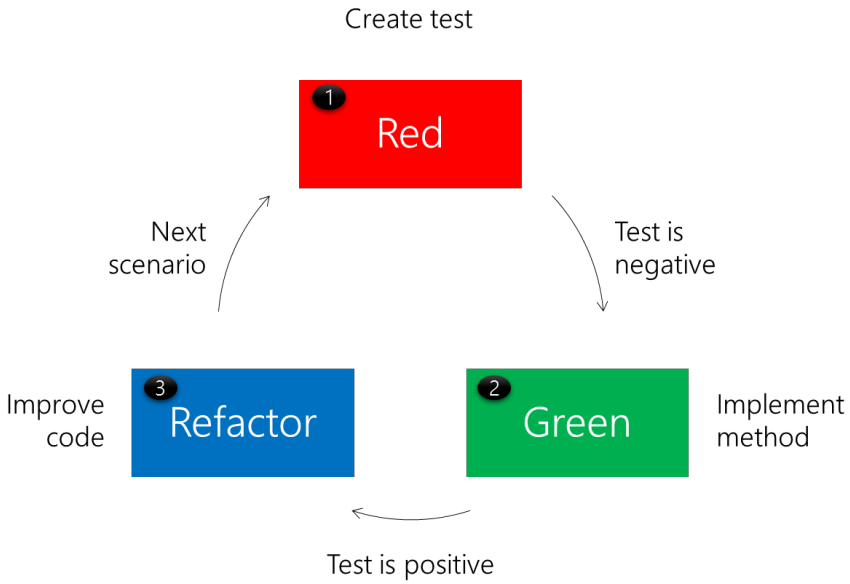
- How to ensure that software is robust? We need tests!
- But software complexity usually grows exponentially

- TDD tries to give us a plan for automated tests
- In the end our software is able to inform us about bugs before we experience them by running the application
- The basic concept is to get some rapid feedback during development
- The risk of change is controlled by having a sufficient number of tests
- We are able to detect problems in the specification

11.7 TDD cycle

- A TDD cycle consists of three phases
  1. Red
  2. Green
  3. Refactor
- In the first phase we ensure that everything is compiling, but the test is failing (since the method just returns a dummy value)
- Now we try to create an implementation that makes the test succeed
- In the third phase we improve the implementation of the method
- Any scenario that needs to be covered by the method has to be tested

11.8 TDD cycle

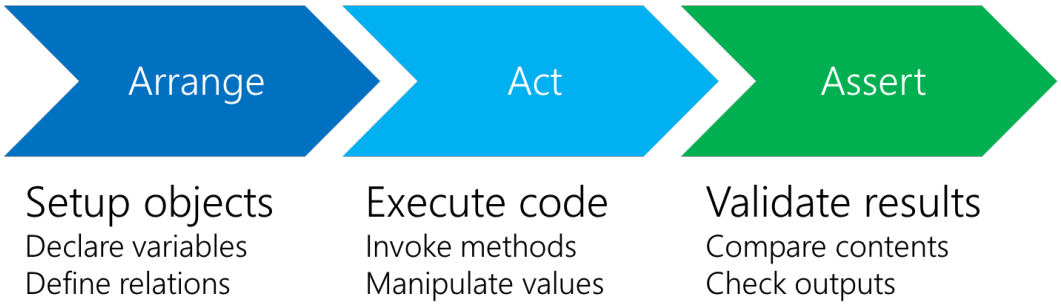


11.9 Red-Green-Refactor

- The red phase is key, since it tries to ensure that the test is bug-free

- One should first see the test failing (if it should) before succeeding
- So overall the process is
  - Add a test and run all tests (new one should fail)
  - Implement the method and run all tests (should be green now)
  - Refactor code and repeat the test (should still be green)
- The test itself should be as simple as possible
- No logic, and following a certain pattern

11.10 Test structure



11.11 Remarks

- Every test consists of creating a test class, performing some setup, invoking the test method and a final cleanup step
- TDD is an important part of any agile development process
- The **KISS** (Keep It Simple Stupid) and **YAGNI** (You Aren't Gonna Need It) principles are usually followed
- This means that small, extensible units are build that only have one responsibility
- The focus lies on the desired job (project goal)

11.12 Shortcomings

- UI, any external resources (databases, filesystems, network, ...) and others are hard to test (require functional tests)
- Writing tests is time-consuming
- Blind spots are more likely if writing the tests is not delegated
- Integration and compliance testing might be reduced due to a false sense of security

- The tests need to be maintained as well
- Big changes in the architecture might result in a time-consuming update of all tests

11.13 Code refactoring

- In short, refactoring is a

" disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. "

- Therefore refactoring should improve code readability by reducing complexity
- Also the code should be made maintenance friendly, yet extensible

11.14 Refactoring strategies

- Refactor when adding a routine
- Refactor when adding a class
- Refactor when fixing a bug
- Target error-prone modules
- Target high-complexity modules
- Improve the parts that are touched
- Define an interface between clean and ugly code

11.15 Reasons to refactor (1)

- Code is duplicated
- Routine is loo long
- A loop is too long or too deeply nested
- A class has poor cohesion
- A class interface does not provide a consistent level of abstraction
- A parameter list has too many parameters
- Changes within a class tend to be compartmentalized
- Changes require parallel modifications to multiple classes

11.16 Reasons to refactor (2)

- Inheritance hierarchies have to be modified in parallel
- Case statements have to be modified in parallel
- Related data items that are used together are not organized into classes
- A routine uses more features of another class than its own class
- A primitive data type is overloaded
- A class doesn't do very much
- A chain of routines passes tramp data
- A middleman object isn't doing anything

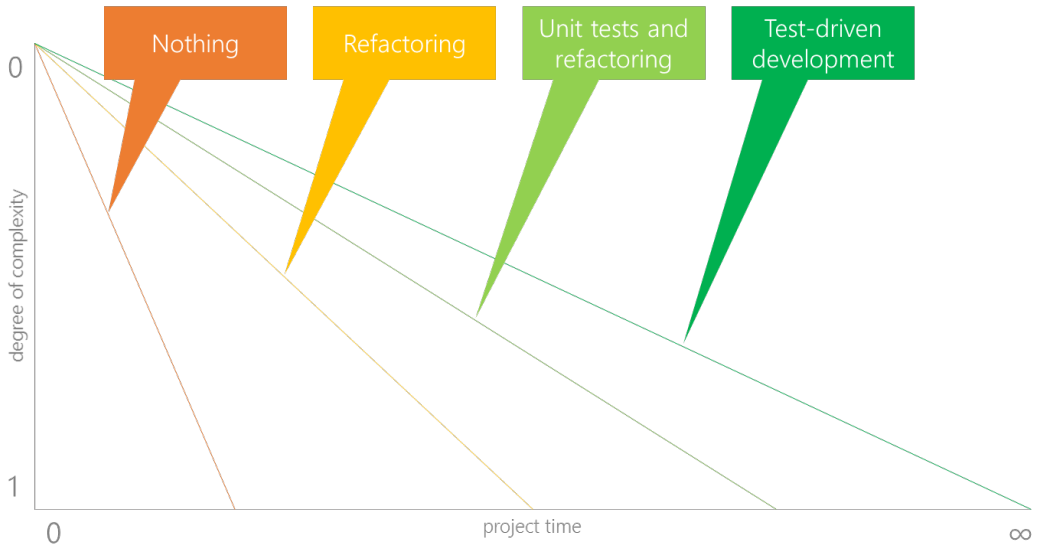
11.17 Reasons to refactor (3)

- One class is overly intimate with another
- A routine has a poor name
- Data members are public
- A subclass uses only a small percentage of its parents' routines
- Comments are used to explain difficult code
- Global variables are used
- A routine uses setup code before a routine call or takedown code after a routine call

11.18 Kinds of refactorings

- Data-Level (e.g. hide class fields)
- Statement-Level (e.g. simplify ifs)
- Routine-Level (e.g. extract method)
- Class implementation (e.g. value to reference)
- Class interface (e.g. SRP)
- System-Level (e.g. factory pattern)

11.19 Refactoring and TDD



11.20 Data-Level refactorings

- Replace a magic number with a named constant
- Rename a variable with a clearer or more informative name
- Move an expression inline or replace an expression with a routine
- Introduce an intermediate variable
- Convert a multi-use variable to multiple single-use variables
- Use a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
- Convert a set of type codes to a class or an enumeration
- Change an array to an object / encapsulate a collection

11.21 Statement-Level refactorings

- Decompose a Boolean expression
- Move a complex Boolean expression into a well-named Boolean function
- Consolidate fragments that are duplicated within different parts of a conditional
- Use break or return instead of a loop control variable
- Return as soon as you know the answer instead of assigning a return value within nested if-then-else statements

- Replace conditionals with polymorphism
- Create and use null objects instead of testing for null values

11.22 Routine-Level refactorings

- Extract routine / extract method
- Move a routine's code inline
- Convert a long routine to a class
- Substitute a simple algorithm for a complex algorithm
- Add or remove a parameter
- Separate query operations from modification operations
- Combine similar routines by parametrization
- Separate routines whose behavior depends on parameters passed in
- Pass a whole object rather than specific fields or vice versa

11.23 Class implementation refactorings

- Change value objects to reference objects
- Change reference objects to value objects
- Replace virtual routines with data initialization
- Change member routine or data placement
- Extract specialized code into a subclass
- Combine similar code into a superclass

11.24 Class interface refactorings

- Move a routine to another class
- Convert one class to two or vice versa
- Hide a delegate or remove a middleman
- Replace inheritance with delegation or vice versa
- Introduce a foreign routine or extension class
- Encapsulate an exposed member variable
- Hide routines that are not supposed to be used outside the class
- Collapse a superclass and subclass if their implementations are very similar

11.25 System-Level refactorings

- Create a defined reference source for data that is beyond our control
- Change unidirectional class associations to bidirectional class associations
- Change bidirectional class associations to unidirectional class associations
- Provide a factory method rather than a simple constructor
- Replace error codes with exceptions or vice versa

11.26 About subroutines

- Historically two kind of operations have been established:
  1. A function (does some computation and returns the result)
  2. A procedure (modifies something and has no result)
- Now we mostly talk about methods (implemented operations of a class)
- There are several questions concerning these operations and their parameters
- Specifically when to extract methods, how to name them and how to structure output and input parameters

11.27 Why another routine?

- Reduce complexity
- Avoid duplicate code
- Support subclassing
- Hide sequences or pointer operations
- Improve portability
- Simplify complicated boolean tests
- Improve performance
- However: **NOT** to ensure that all routines are small!

11.28 Naming routines

- Describe what the routine does
- Avoid meaningless verbs
- Don't use numbers for differentiation
- Make routine names as long as necessary (for variables 9-15 chars)
- For a procedure: Use a strong verb followed by an object
- For a function: Use a description of the return value
- Otherwise use opposites precisely (open/close, add/remove, create/destroy)
- Establish connections

11.29 Routine parameters

- Parameters should always be as general as possible
- The opposite is the type of the return value
- This type should be as specific as possible
- We call incoming parameters contra-variant
- Outgoing parameters (return values) are co-variant
- Such a style increases flexibility by allowing methods to be used with more types
- Also the return value is then a lot more useful

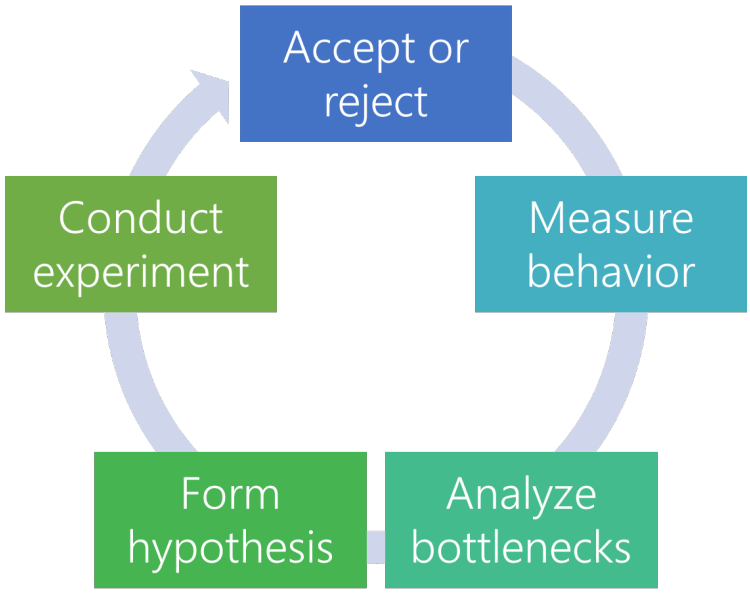
11.30 Using parameters

- Follow the order: Input-Modify-Output
- Consider creating own input and output keywords if possible
- Otherwise create or use conventions like *in**, *out** for names
- Use all parameters
- Put status or error variables last
- Don't use routine parameters as working variables
- Limit routine parameters to max. 7
- Use named parameters if possible

11.31 Code optimization

- Not trivial, since code should be readable and follow our conventions
- Nevertheless sometimes parts of the application are performance critical
- Problem: Most optimizations should have been integrated in the design
- But "*premature performance optimization is the root of all evil*"
- Only solution: Try to maximize performance and change design if still not good enough

11.32 Performance analysis



11.33 Common techniques

- Substitute table lookups for complicated logic
- Jam loops
- Use integer instead of floating point variables when possible
- Initialize data at compile time
- Use constants of the correct type
- Precompute results
- Eliminate common subexpressions
- Translate key routines to a low-level language

11.34 Quick improvements (1)

- Order tests (switch-case, if-else) by frequency
- Compare performance of similar logic structures
- Use lazy evaluation
- Unswitch loops that contain if tests
- Unroll loops
- Minimize work performed in loops
- Put the busiest loop on the inside of nested loops

11.35 Quick improvements (2)

- Change multi-dimensional to one-dimensional array
- Minimize array references
- Augment data types with indices
- Cache frequently used variables
- Exploit algebraic identities
- Reduce strength in logical and mathematical expressions
- Rewrite routines inline

11.36 References

- CodeProject: Object Oriented Design Principles (`http://www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles`)
- Introduction to Test Driven Development (`http://www.agiledata.org/essays/tdd.html`)
- Wikipedia: Test-driven development (`http://en.wikipedia.org/wiki/Test-driven_development`)
- Martin Fowler: Refactoring (`http://refactoring.com/`)
- Slideshare: Code tuning (`http://www.slideshare.net/bgtraghu/code-tuning`)
- Sourcemaking: Refactoring (`http://sourcemaking.com/refactoring`)

11.37 Literature

- McConnell, Steve (2004). *Design in Construction.*
- Sedgewick, Robert (1984). *Algorithms.*
- Kerievsky, Joshua (2004). *Refactoring to Patterns.*
- Fowler, Martin (1999). *Refactoring: Improving the design of existing code.*
- Weisfeld, Matt (2004). *The Object-Oriented Thought Process.*
- Beck, Kent (2003). *Test-Driven Development by Example.*

Chapter 12

Clean code

12.1 Introduction

- Clean code has been made popular by Robert C. Martin
- In fact clean code tries to use a wide range of techniques to ensure
  - stable and robust programs
  - a short development time for extensions
  - fewer bugs and less maintenance cycles
- We already learned a lot of techniques that are used with clean code
- Clean code is also driven by coding conventions and custom conventions

12.2 Grades

- The Clean Code Developer (CCD) initiative tries to embody all concepts
- The grade of a developer represents his skill / pursuit of these rules
- In total there are six grades:
  1. red
  2. orange
  3. yellow
  4. green
  5. blue
  6. white

12.3 Red

- Don't Repeat Yourself (DRY)
- Keep It Simple, Stupid (KISS)

- Favor Composition over Inheritance (FCoI)
- Using a version control system (VCS, e.g. git)
- Be cautious with optimizations
- Perform extensive root cause analysis
- Apply simple code refactoring methods

12.4 Orange

- Use a Single Level of Abstraction (SLA)
- Follow the SRP (see SOLID principles)
- Define and apply source code conventions
- Setup automatic integration tests
- Create a system for tracking issues
- Structure by Separation Of Concerns (SOC)
- Actively participate in code reviews and read books

12.5 Yellow

- Use three more SOLID principles:
  1. Interface Segregation Principle (ISP)
  2. Dependency Inversion Principle (DIP)
  3. Liskov Substitution Principle (LSP)
- Perform automatic unit tests together with a code coverage analysis
- Create mockups for advanced testing
- Follow the Information Hiding Principle (IHP)

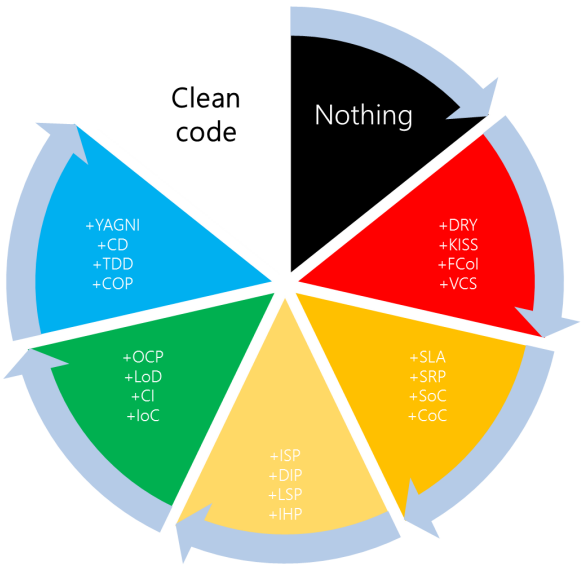
12.6 Green

- Open Close Principle (OCP) (now SOLID is fully applied)
- Use a continuous integration system
- Follow the law of demeter
- Integrate an Inversion of Control (IoC) container
- Evaluate code metrics
- Let classes follow the *tell, don't ask* principle

12.7 Blue

- Continuous delivery in place
- Test-driven development by test first
- Iterative development (small, very robust, yet extensible components)
- You Ain't Gonna Need It (YAGNI)
- Structuring applications in pieces, so called modules
- A class is then a piece of a module
- This modularity encourages parallel development

12.8 White



12.9 Information hiding

- Sometimes called encapsulation (*controversial*)
- The storage layout should be hidden and cannot be accessed outside
- Main advantage: Changing the layout will not change anything else
- Any kind of method that is implementation specific should be hidden
- Encapsulation is also a technique for providing a stable interface between multiple systems
- Some people consider inheritance to break encapsulation

12.10 Composition over inheritance

- Sometimes called *Composite Reuse Principle* and describes a technique for achieving polymorphic behavior
- Classes should references to other classes, that implement the desired functionality
- Therefore any functionality is outsourced and SRP is easily possible
- The ultimate goal is higher flexibility
- In other words, **has a** can be better than an **is a** relationship
- However, all methods need to be specified, not just a subset

12.11 Example

```
class GameObject
{
    readonly IVisible _v;
    readonly IUpdatable _u;
    readonly ICollidable _c;

    public GameObject(IVisible v, IUpdatable u, ICollidable c)
    {
        _v = v;
        _u = u;
        _c = c;
    }

    public void Update()
    {
        _u.Update();
    }

    public void Draw()
    {
        _v.Draw();
    }

    public void Collide()
    {
        _c.Collide();
    }
}
interface IVisible
{
    void Draw();
}
class Invisible : IVisible
{
    public void Draw()
    {
    }
```

```
}
class Visible : IVisible
{
    public void Draw()
    {
        /* draw model */
    }
}
interface ICollidable
{
    void Collide();
}
class Solid : ICollidable
{
    public void Collide()
    {
        /* check collisions with object and react */
    }
}
class NotSolid : ICollidable
{
    public void Collide()
    {
    }
}
interface IUpdatable
{
    void Update();
}
class Movable : IUpdatable
{
    public void Update()
    {
        /* move object */
    }
}
class NotMovable : IUpdatable
{
    public void Update()
    {
    }
}
class Player : GameObject
{
    public Player() : base(new Visible(), new Movable(), new Solid())
    {

    }
}
class Smoke : GameObject
{
    public Smoke() : base(new Visible(), new Movable(), new NotSolid())
    {

    }
}
```

12.12 Law of Demeter

- A specific case of loose coupling (design guideline)
- Each class should have only limited knowledge about other classes
- Only *closely* related classes should be known
- No communication with unrelated classes
- More strictly a class should never call the method of an object contained in another class, i.e. a.b.c() is not allowed, but a.b() is
- In general a method can only call methods from the current class, one of the parameters or a method of a global variable (application or class)

12.13 Single Level of Abstraction

- Readability is improved by clear structures (e.g. formatting)
- An important detail is the level of abstraction of a block
- The class name is one level, the public methods another
- It is important to never mix levels of abstraction
- Example: Bit-wise operations should not be mixed with method calls
- Reason: Reader decides how deep to dive into the code

12.14 Code coverage

- Code coverage describes how many possible paths are tested
- There are several kinds of coverages:
  - Function coverage (is the function tested?)
  - Statement coverage (is the statement called?)
  - Condition coverage (is any possible value included?)
  - State coverage (have all possible states been reached?)
- A high indicator (higher 80%) is usually a sign for a well-tested code

12.15 Code metrics

- Measuring the quality of software is usually highly subjective
- Code metrics give numbers that can be used to estimate the quality
  - Maintainability index (0 to 100, high is better)
  - Cyclomatic complexity (number of different paths)
  - Depth of inheritance (lower is better)
  - Class coupling (lower is better)
  - Lines of code (or number of instructions)
- Cyclomatic complexity is the most important measurement

12.16 IoC container

- Task: Assembling components from different projects into one app
- Problem: How to do the wiring? Loose coupling?!
- Of course inversion of control has to be used (from the DIP) with factory
- Two choices for such a factory (called container):
  1. Service Locator (SL)
  2. Dependency Injection (DI)
- The SL has to be called explicitly, DI is an implicit construction (target classes don't need to know how the dependency will be resolved)

12.17 Continuous delivery

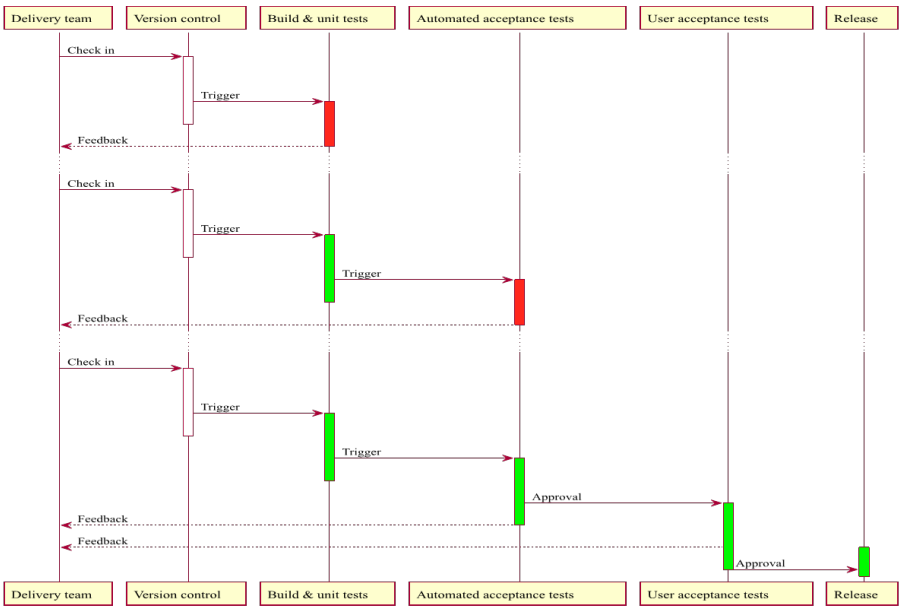


Figure 12.1: Continuous Delivery

12.18 References

- CodeProject: Design Pattern FAQ (Part 1) (`http://www.codeproject.com/Articles/28309/Design-pattern-FAQ-Part-1-Training`)
- CodeProject: DIP in depth (`http://www.codeproject.com/Articles/538536/A-curry-of-Dependency-Inversion-Principle-DIP-Inve`)
- Martin Fowler about UI architectures (`http://www.martinfowler.com/eaaDev/uiArchs.html`)
- Martin Fowler about Dependency Injection (`http://martinfowler.com/articles/injection.html`)
- MSDN: Code metrics (`http://msdn.microsoft.com/en-us/library/bb385914.aspx`)
- Wikipedia: Code coverage (`http://en.wikipedia.org/wiki/Code_coverage`)
- The Clean Code Developer initiative (`http://www.clean-code-developer.de/`)

12.19 Literature

- Shalloway, Alan; Trott, James (2002). *Design Patterns Explained.*
- Humble, Jez; Farley, David (2010). *Continuous delivery : reliable software releases through build, test, and deployment automation.*

- Martin, Robert (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.*
- Evans, Eric (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.*
- Beck, Kent (2002). *Test Driven Development. By Example.*