Laboratorio di Sistemi Software

Design Patterns 3

Luca Padovani (A-L)
Riccardo Solmi (M-Z)

Indice degli argomenti

- **Tipi di Design Patterns**
  - Creazionali, strutturali, comportamentali
- **Design Patterns**
  - Factory Method, Strategy, State, Decorator
  - Composite, Iterator, Visitor
  - Template Method, Abstract Factory, Builder, Singleton
  - Proxy, Adapter, Bridge, Facade
  - Mediator, Observer, Chain of Responsibility
  - Command, Prototype

© 2002-2003 Riccardo Solmi

Bibliografia

- *UML*
  - *Linguaggio per modellare un Sistema Software.*
- *Design Patterns – Elements of Reusable Object-Oriented Software*
  - Catalogo di patterns di progettazione
- *Refactoring – Improving the Design of Existing Code*
  - Catalogo di operazioni di Refactoring
- *Eclipse IDE + UML Plugin*
  - Ambiente di programmazione che supporta refactoring e UML.
- **Trovate tutto nelle pagine web relative a questo corso:**
  - http://www.cs.unibo.it/~solmi/teaching/labss_2002-2003.html
  - http://www.cs.unibo.it/~lpadovan/didattica.html

© 2002-2003 Riccardo Solmi

Ricevimento

- *Metodo 1: il newsgroup*
  - Avete a disposizione un newsgroup: unibo.cs.informatica.paradigmiprogrammazione
  - Utilizzatelo il più possibile; se nessuno risponde in due-tre giorni, risponderà uno dei docenti
  - Valuteremo anche la partecipazione al newsgroup
- *Metodo 2: subito dopo le lezioni*
  - siamo a vostra disposizione per chiarimenti
  - in aula e alla lavagna, in modo da rispondere a tutti
- *Metodo 3: ricevimento su appuntamento*
  - Scrivete a lpadovan@cs.unibo.it (A-L) o a solmi@cs.unibo.it (M-Z)

© 2002-2003 Riccardo Solmi

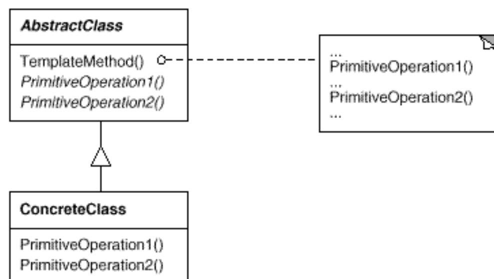
Template Method

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- **Applicability**
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication
  - to control subclasses extensions

© 2002-2003 Riccardo Solmi

Template Method /2

- **Structure**



- **Participants**
  - **AbstractClass**
  - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
  - **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

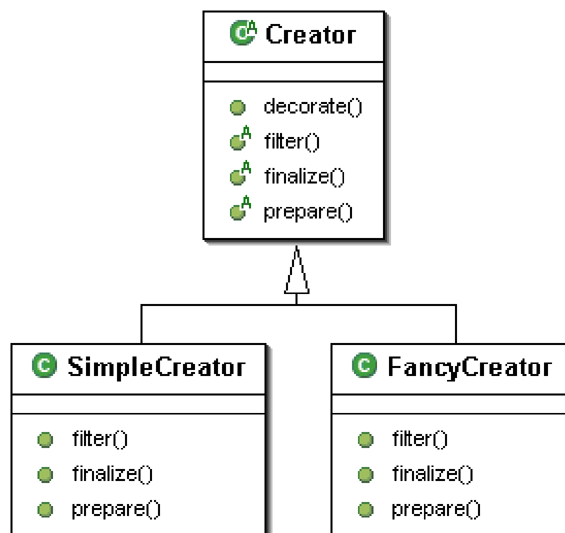
© 2002-2003 Riccardo Solmi

Template Method /3

- **Collaborations**
  - ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm
- **Consequences**
  - Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you"
  - It's important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding

© 2002-2003 Riccardo Solmi

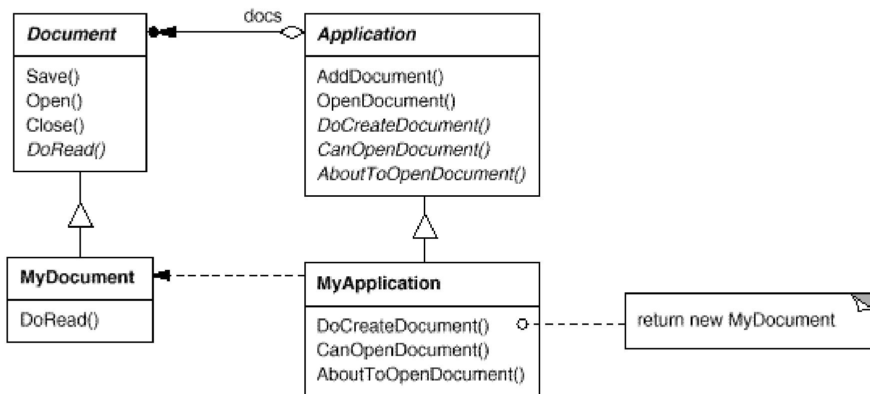
Template Method example 1



© 2002-2003 Riccardo Solmi

Template Method example 2

- **Application framework that provides Application and Document classes**



© 2002-2003 Riccardo Solmi

Template Method questions

- **The Template Method relies on inheritance. Would it be possible to get the same functionality of a Template Method, using object composition? What would some of the tradeoffs be?**

© 2002-2003 Riccardo Solmi

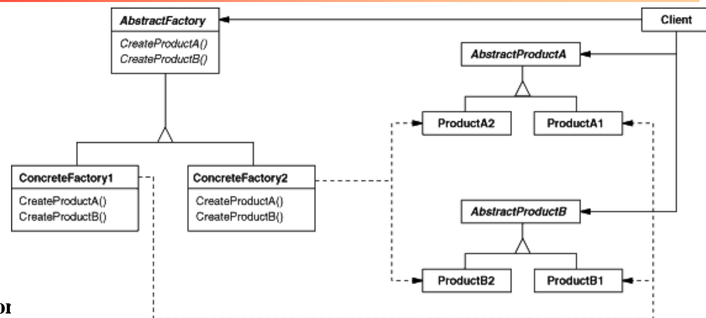
Abstract Factory

- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- **Applicability**
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

© 2002-2003 Riccardo Solmi

Abstract Factory /2

- **Structure**



- **Participants**
  - **AbstractFactor**
  - declares an interface for operations that create abstract product objects.
  - **ConcreteFactory**
  - implements the operations to create concrete product objects.
  - **AbstractProduct**
  - declares an interface for a type of product object.
  - **ConcreteProduct**
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
  - **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes

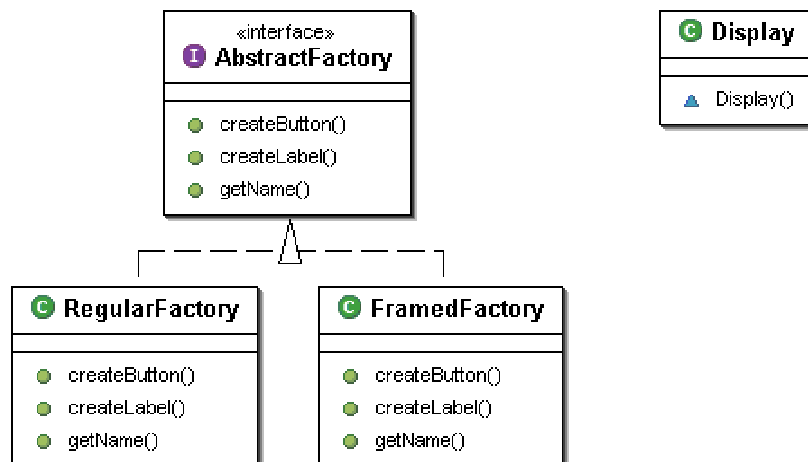
© 2002-2003 Riccardo Solmi

Abstract Factory /3

- **Collaborations**
  - Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation.
  - AbstractFactory defers creation of product objects to its ConcreteFactory subclass
- **Consequences**
  - It isolates concrete classes
  - It makes exchanging product families easy
  - It promotes consistency among products
  - Supporting new kinds of products is difficult

© 2002-2003 Riccardo Solmi

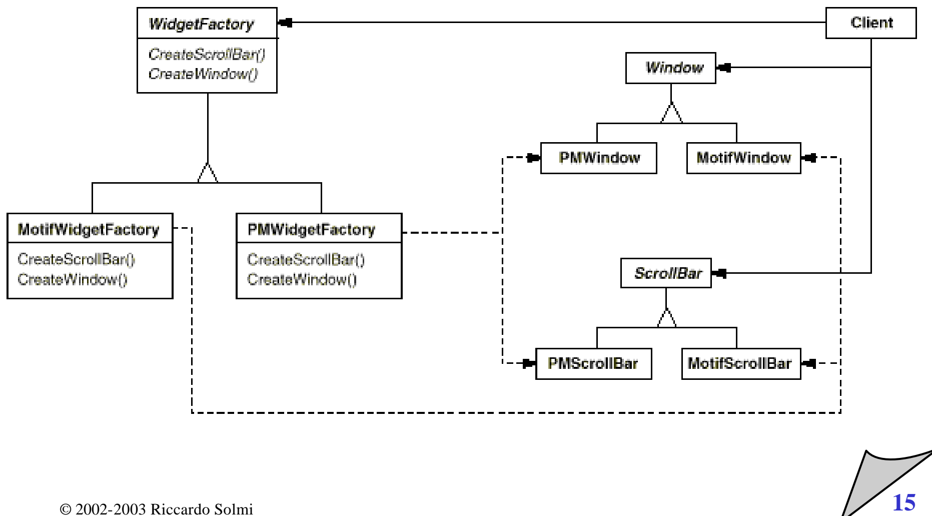
Abstract Factory example 1



© 2002-2003 Riccardo Solmi

Abstract Factory example 2

- User interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager



© 2002-2003 Riccardo Solmi

Abstract Factory questions

- In the Implementation section of this pattern, the authors discuss the idea of *defining extensible factories*. Since an Abstract Factory is composed of Factory Methods, and each Factory Method has only one signature, does this mean that the Factory Method can only create an object in one way?

© 2002-2003 Riccardo Solmi

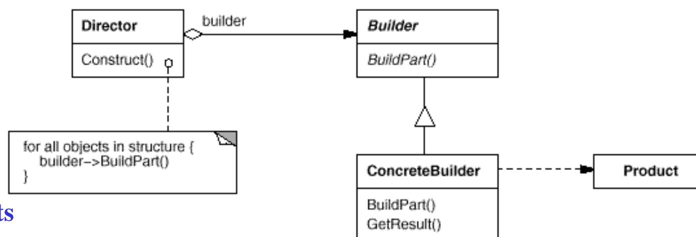
Builder

- **Intent**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- **Applicability**
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - the construction process must allow different representations for the object that's constructed.

© 2002-2003 Riccardo Solmi

Builder /2

- **Structure**



- **Participants**
- **Builder**
- specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder**
- constructs and assembles parts of the product by implementing the Builder interface.
- defines and keeps track of the representation it creates.
- provides an interface for retrieving the product
- **Director**
- constructs an object using the Builder interface.
- **Product**
- represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

© 2002-2003 Riccardo Solmi

Builder /3

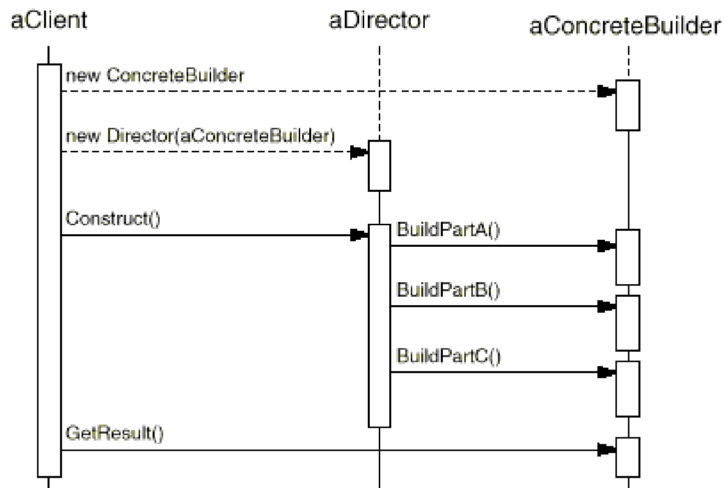
- **Collaborations**
  - Client creates Director object and configures it with desired Builder object.
  - Director notifies the builder whenever a part of the product should be built.
  - Builder handles requests from the director and adds parts to the product.
  - The client retrieves the product from the builder.

- **Consequences**
  - It lets you vary a product's internal representation
  - It isolates code for construction and representation
  - It gives you finer control over the construction process

© 2002-2003 Riccardo Solmi

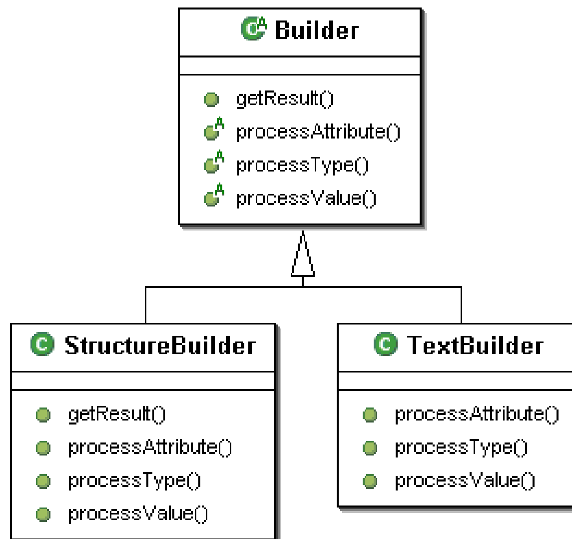
Builder /4

- **The following interaction diagram illustrates how Builder and Director cooperate with a client.**



© 2002-2003 Riccardo Solmi

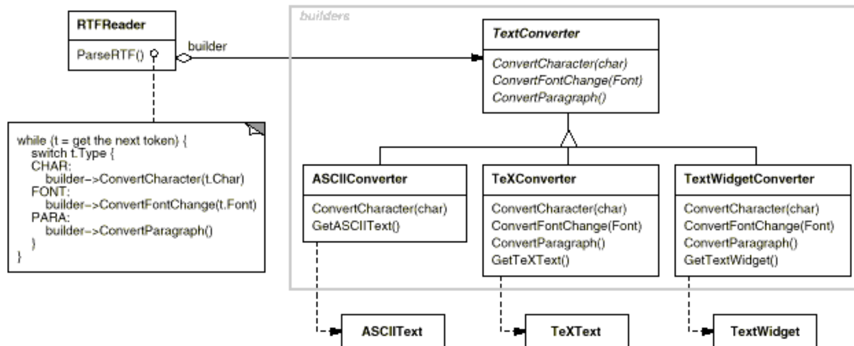
Builder example 1



© 2002-2003 Riccardo Solmi

Builder example 2

- **A reader for the RTF document exchange format should be able to convert RTF to many text formats. The problem is that the number of possible conversions is open-ended**



© 2002-2003 Riccardo Solmi

Builder questions

- **Like the Abstract Factory pattern, the Builder pattern requires that you define an interface, which will be used by clients to create complex objects in pieces. How does the Builder pattern allow one to add new methods to the Builder's interface, without having to change each and every sub-class of the Builder?**

© 2002-2003 Riccardo Solmi

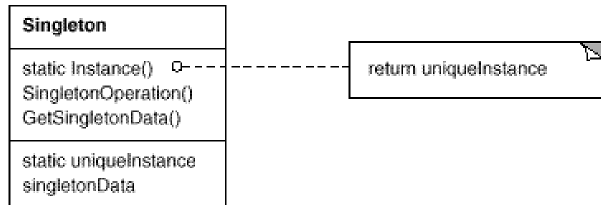
Singleton

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

© 2002-2003 Riccardo Solmi

Singleton /2

- **Structure**



- **Participants**
- **Singleton**
- defines an Instance operation that lets clients access its unique instance. Instance is a class operation (that is, a static method in Java and a static member function in C++).
- may be responsible for creating its own unique instance.

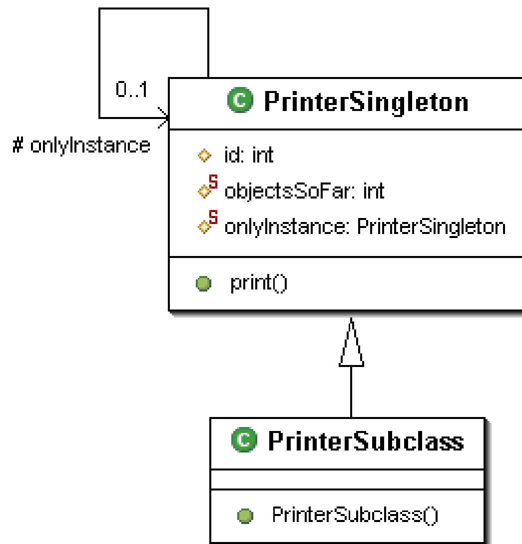
© 2002-2003 Riccardo Solmi

Singleton /3

- **Collaborations**
  - Clients access a Singleton instance solely through Singleton's Instance operation.
- **Consequences**
  - Controlled access to sole instance
  - Reduced name space
  - Permits refinement of operations and representation
  - Permits a variable number of instances
  - More flexible than class operations

© 2002-2003 Riccardo Solmi

Singleton example 1



© 2002-2003 Riccardo Solmi

Singleton example 2

- It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

© 2002-2003 Riccardo Solmi

Singleton questions

- **The Singleton pattern is often paired with the Abstract Factory pattern. What other creational or non-creational patterns would you use with the Singleton pattern?**

© 2002-2003 Riccardo Solmi

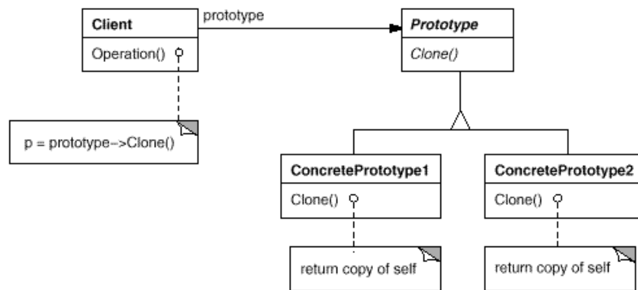
Prototype

- **Intent**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- **Applicability**
  - when a system should be independent of how its products are created, composed, and represented; *and*
  - when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
  - to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
  - when instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

© 2002-2003 Riccardo Solmi

Prototype /2

- **Structure**



- **Participants**
  - **Prototype**
  - declares an interface for cloning itself.
  - **ConcretePrototype**
  - implements an operation for cloning itself.

•**Client**

•creates a new object by asking a prototype to clone itself.

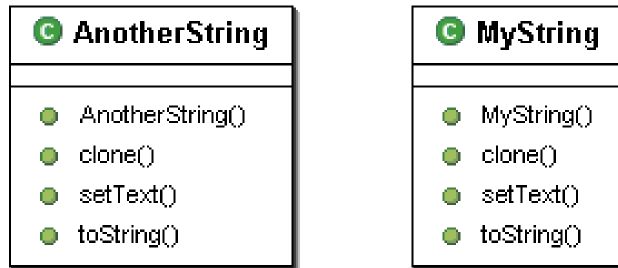
© 2002-2003 Riccardo Solmi

Prototype /3

- **Collaborations**
  - A client asks a prototype to clone itself.
- **Consequences**
  - hides the concrete product classes from the client
  - Adding and removing products at run-time
  - Specifying new objects by varying values
  - Specifying new objects by varying structure
  - Reduced subclassing
  - Configuring an application with classes dynamically

© 2002-2003 Riccardo Solmi

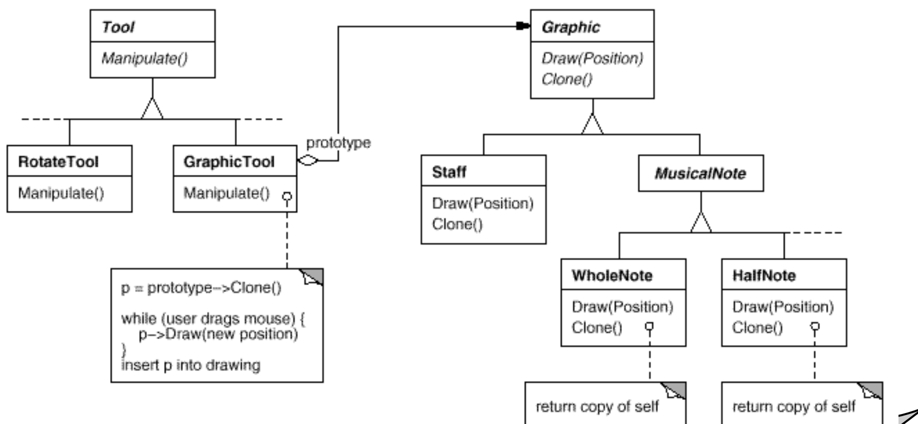
Prototype example 1



© 2002-2003 Riccardo Solmi

Prototype example 2

- **Build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves**



© 2002-2003 Riccardo Solmi

Prototype questions

- **Part 1: When should this creational pattern be used over the other creational patterns?**
- **Part 2: Explain the difference between deep vs. shallow copy**

© 2002-2003 Riccardo Solmi