Design Patterns

Produced by

Eamonn de Leastar

edeleastar@wit.ie

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

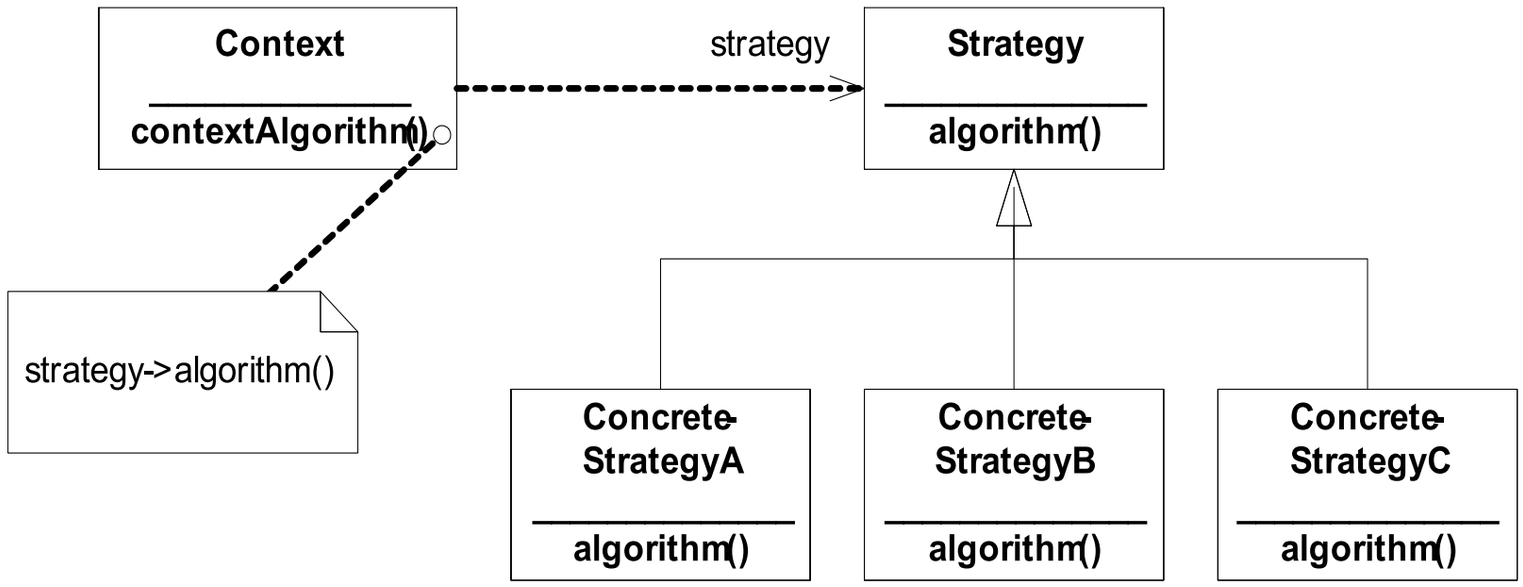
Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Strategy

Design Pattern

Summary



- Have the class that uses the algorithm contain a reference to an interface (or abstract class) that has a method specifying a specific algorithm.

- Each implementation of the interface (or derived class) implements the algorithm as appropriate.

Intent

- Define an interface that defines a strategy for performing some operation.
- A family of interchangeable classes, one for each algorithm, implements the interface.
- Also known as: Policy

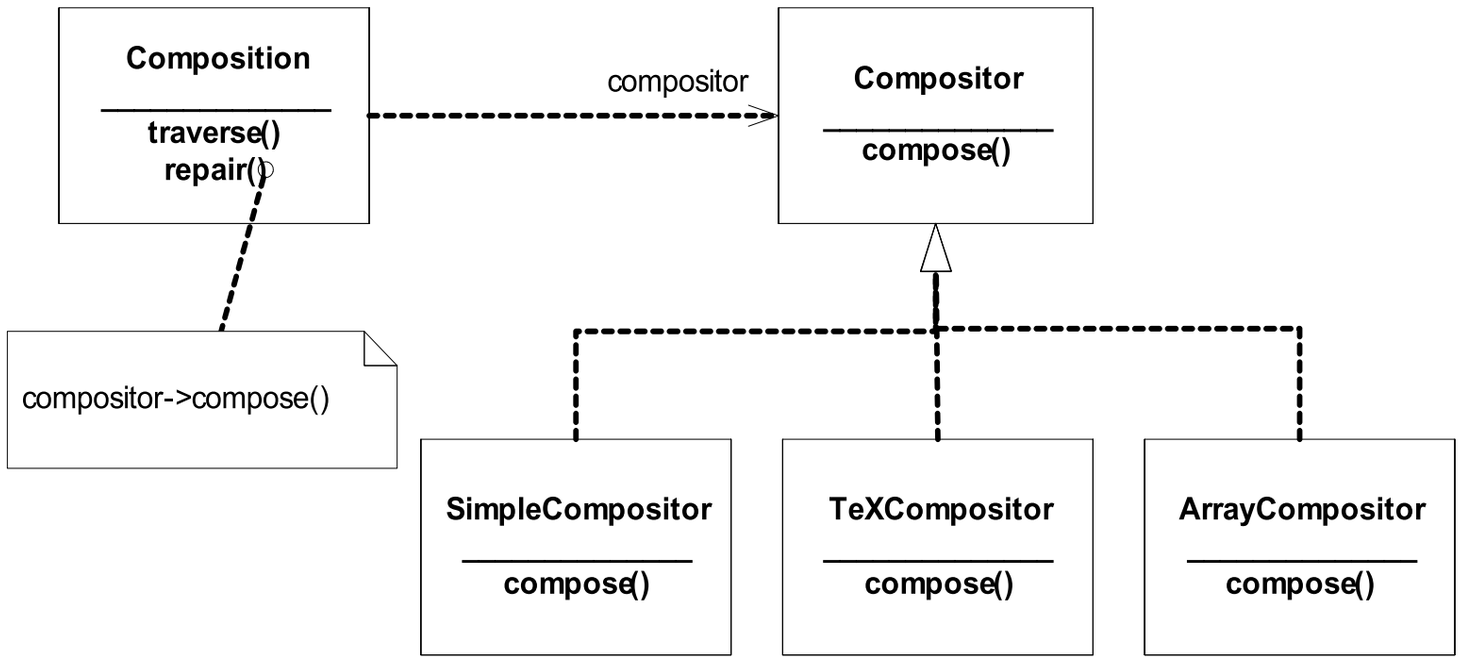
Motivation (1)

- Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:
  - Clients that need line breaking get more complex if they include the line breaking code. That makes clients bigger and harder to maintain, especially if they support multiple line breaking algorithms.
  - Different algorithms will be appropriate at different times. We don't want to support multiple line breaking algorithms if we don't use them all.
  - It's difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client.
- We can avoid these problems by defining classes that encapsulate different line breaking algorithms. An algorithm that's encapsulated in this way is called a strategy

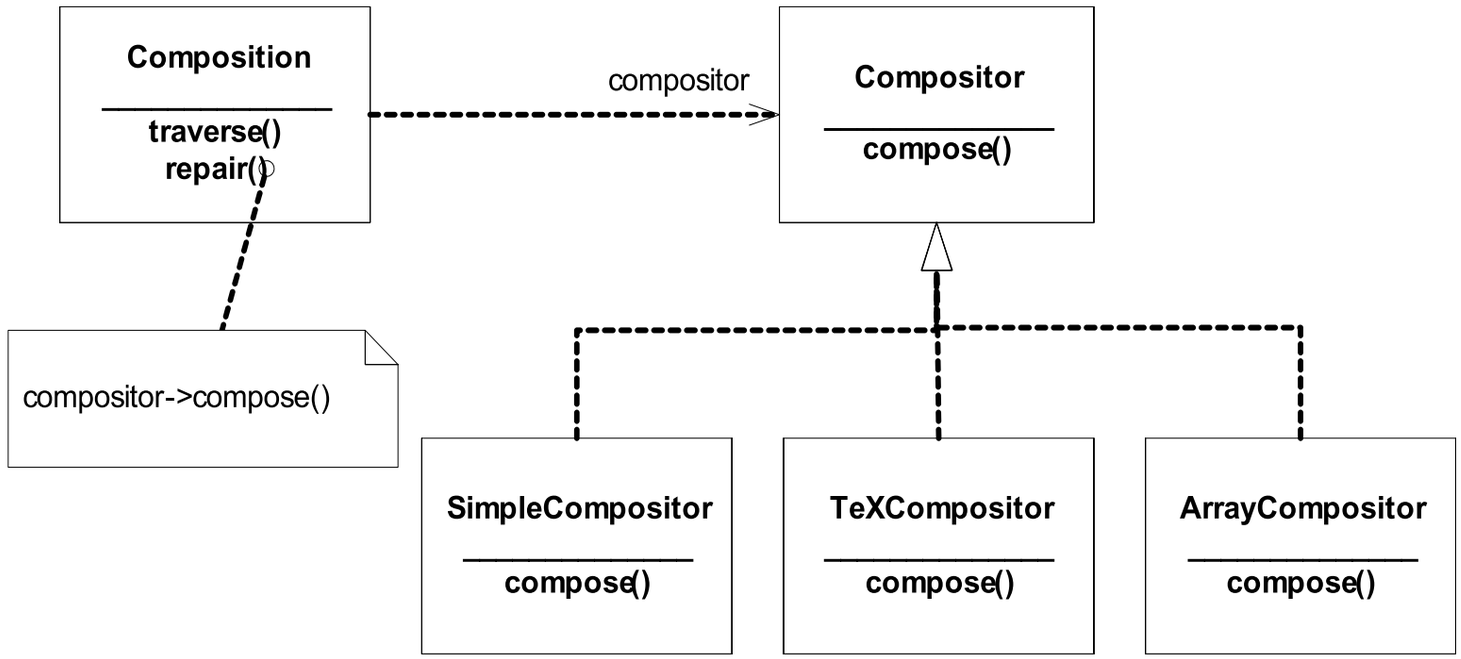
Motivation (2)

Composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer.

Linebreaking strategies aren't implemented by the class Composition. Instead, they are implemented separately by subclasses of the abstract Compositor class. Compositor subclasses implement different strategies:



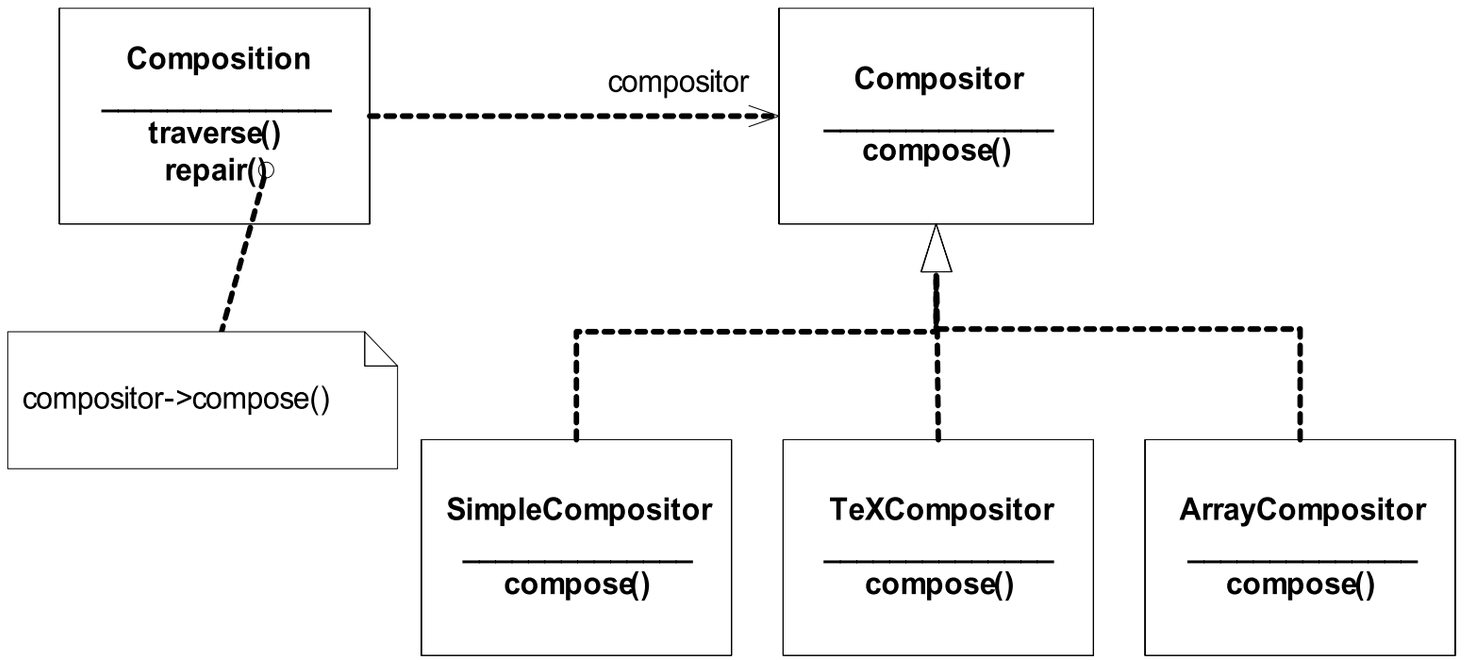
Motivation (2)



- SimpleCompositor implements a simple strategy that determines linebreaks one at a time.
- TeXCompositor implements the TeX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
- ArrayCompositor implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

Motivation (2)

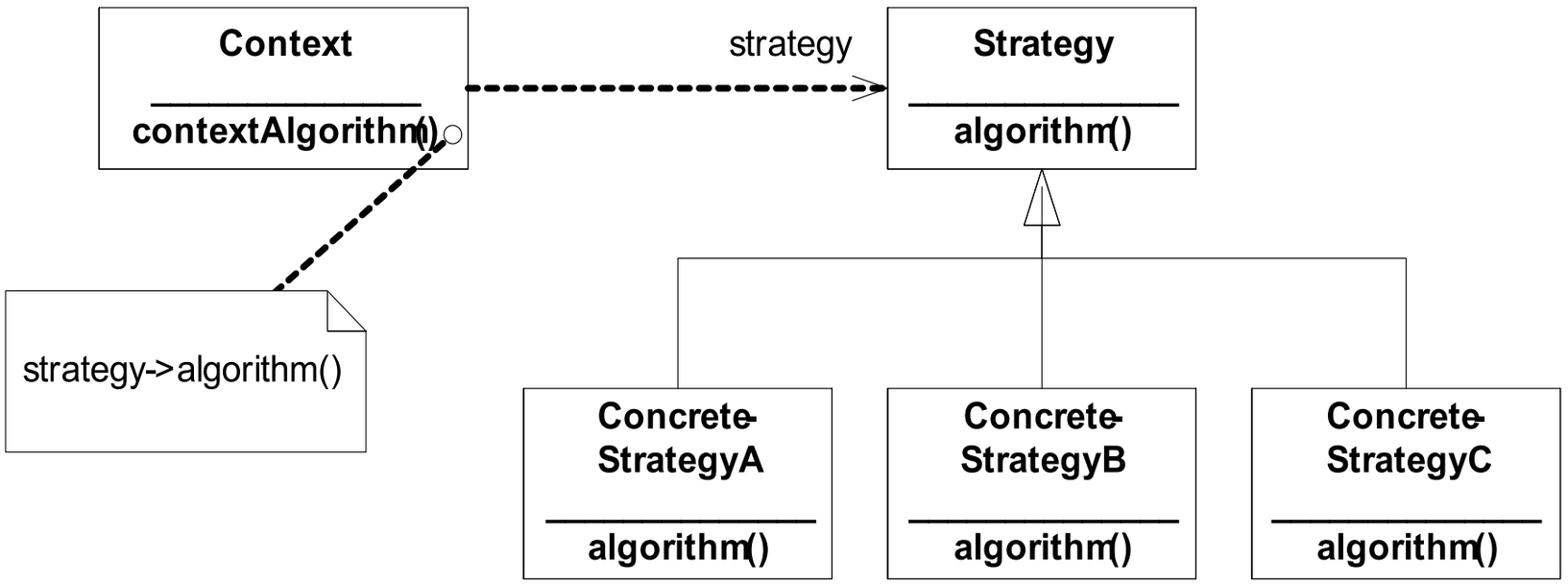
A Composition maintains a reference to a Compositor object. Whenever a Composition reformats its text, it forwards this responsibility to its Compositor object. The client of Composition specifies which Compositor should be used by installing the Compositor it desires into the Composition.



Applicability

- Use the Strategy pattern when
  - many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
  - you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs.
  - an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related

Structure


Context
contextAlgorithm()
strategy
Strategy
algorithm()
strategy->algorithm()
ConcreteStrategyA
algorithm()
ConcreteStrategyB
algorithm()
ConcreteStrategyC
algorithm()

Participants

- Strategy (Compositor)
    - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor)
    - implements the algorithm using the Strategy interface.
- Context (Composition)
- is configured with a ConcreteStrategy object.
    - maintains a reference to a Strategy object.

Collaborations

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.

- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences (1)

- Families of related algorithms : Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse

- An alternative to subclassing : Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

- Strategies eliminate conditional statements: When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.

- A choice of implementations: Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

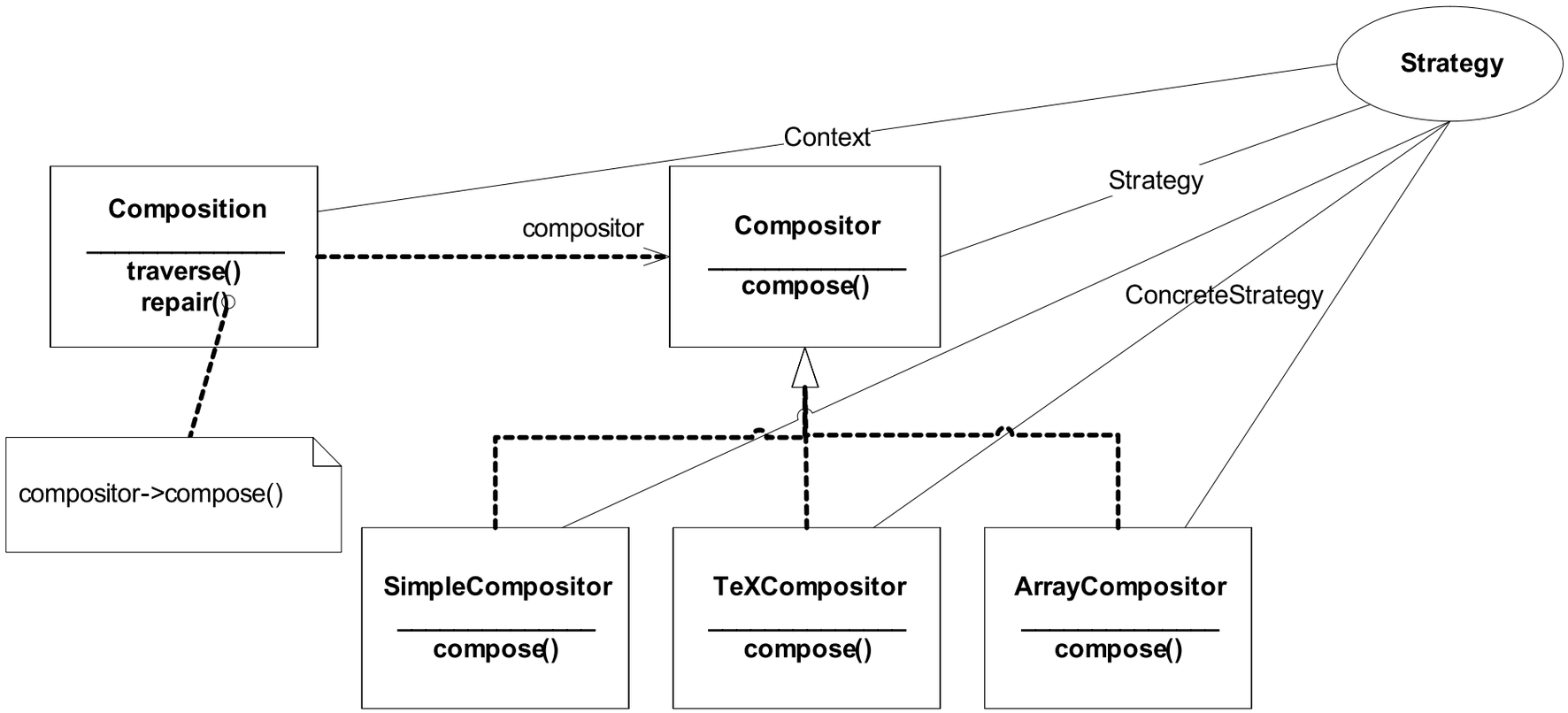
Consequences (2)

- Clients must be aware of different Strategies: client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients

- Communication overhead between Strategy and Context: The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface

- Increased number of objects: Strategies increase the number of objects in an application.

Strategy UML



Strategy vs Template Method

- Both solve the problem of separating a generic algorithm from a detailed context
    - Strategy is like Template Method except in its granularity.
    - Template Method uses inheritance to vary parts of an algorithm.
    - Strategy uses delegation to vary the entire algorithm.
- Template method ensures that the variable parts are extended coherently.
- Template method, however, leaves code prone to “Fragile Base Class” problem.