# Software design: design patterns

- Software development patterns
- Template Method pattern

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

# Software Development Patterns

## A pattern:

- “describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Alexander et al. (1977)

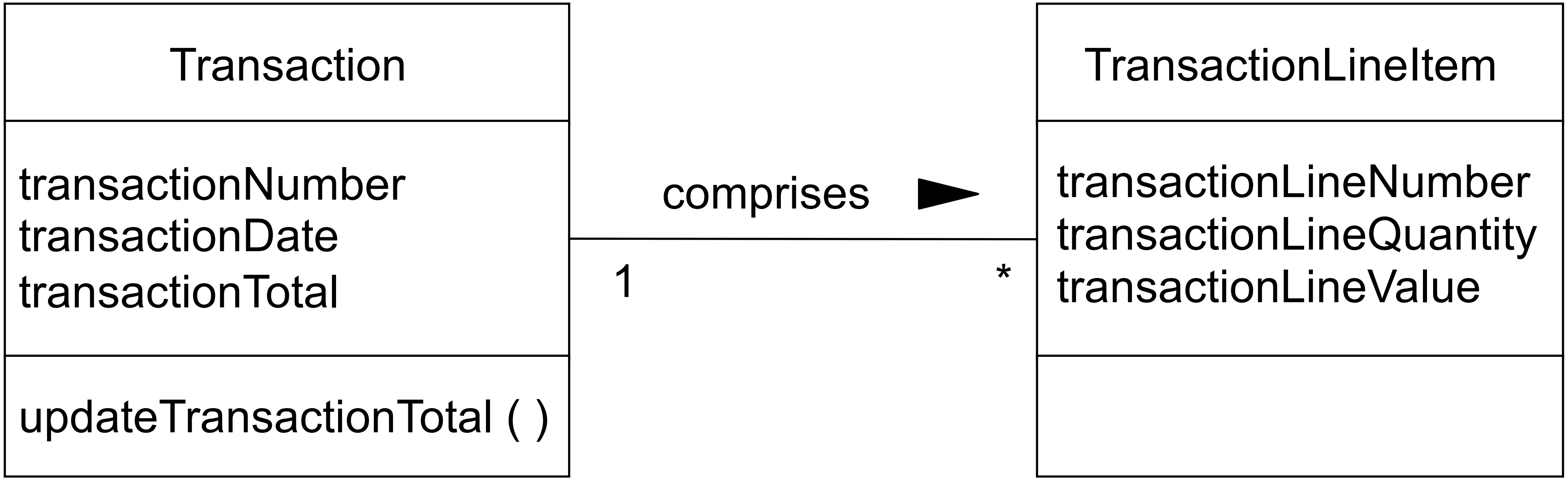
# Software Development Patterns

- A pattern has:
  - A *context* = a set of circumstances or preconditions for the problem to occur
  - *Forces* = the issues that must be addressed
  - A software configuration that resolves the forces

# Software Development Patterns

- Patterns are found at many points in the systems development lifecycle:
  - **Analysis patterns** are groups of concepts useful in modelling requirements (see lecture on Type diagrams)
  - **Architectural patterns** describe the structure of major components of a software system (see lecture next week)
  - **Design patterns** describe the structure and interaction of smaller software components

# Simplest Analysis Pattern



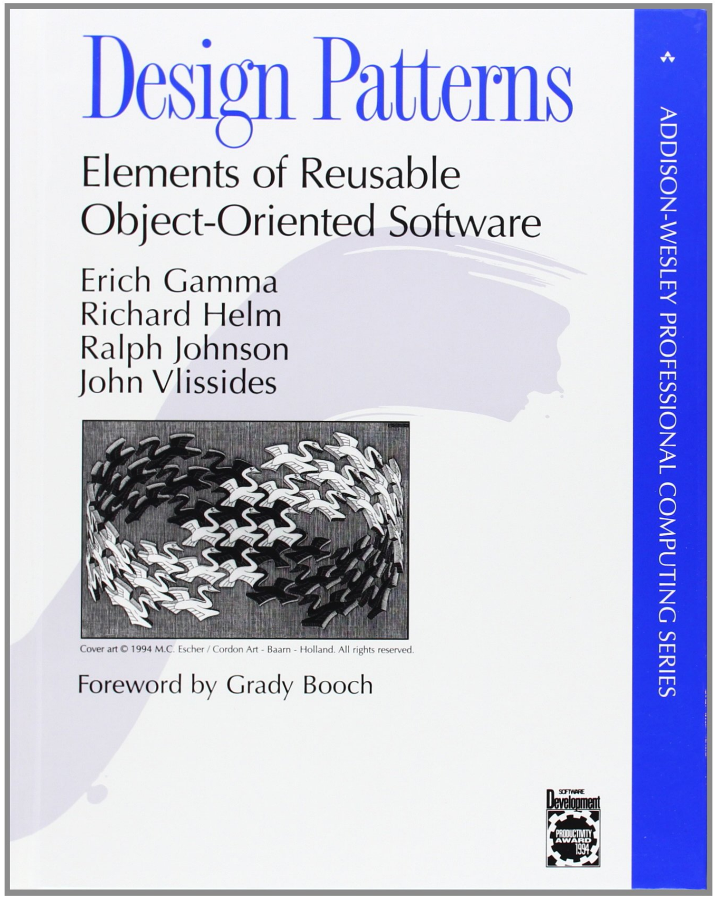
# Template Method pattern

# The Template Method pattern

- 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.

- In object-oriented programming, first a class is created that provides the **basic steps** of an algorithm design.
- These steps are implemented using **abstract** methods.
- Later on, subclasses change the abstract methods to **implement real actions**. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.
- Avoids **duplication** in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses.

# Design Patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- The template method is one of the twenty-three well-known patterns described in the "Gang of Four" book Design Patterns (1994).

# Template Method: Example 1

An abstract class that is common to several games in which players play against the others, but only one is playing at a given time.

```
abstract class Game {

    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    /* A template method : */
    public final void playOneGame(int playersCount)
{
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
```

```
class Monopoly extends Game {

    /* Implementation of necessary concrete methods
*/
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int pla
        // Process one turn
    }
    boolean endOfGame(
        // Return true if gar
        // according to Mor
    }
    void printWinner() {
        // Display who wor
    }
    /* Specific declaration

    // ...
}
```

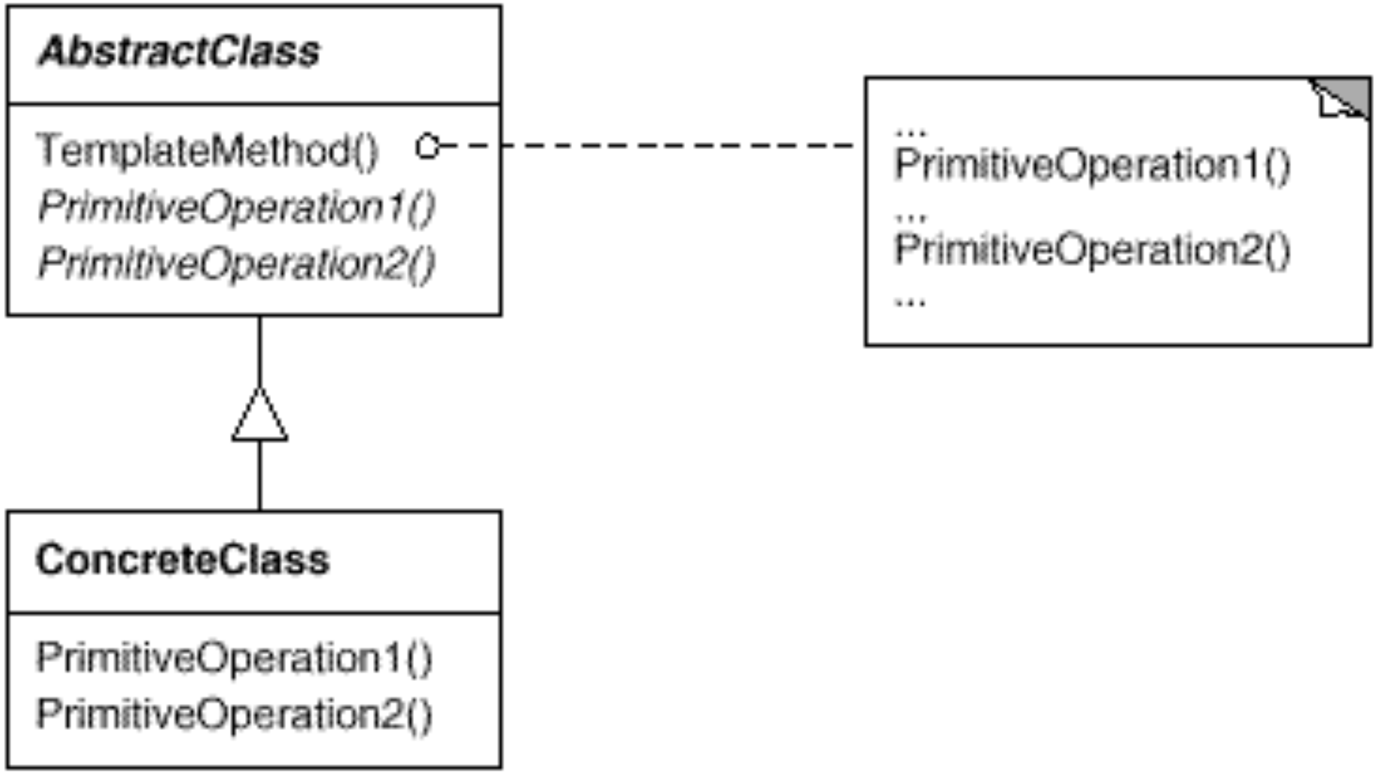
```
class Chess extends Game {

    /* Implementation of necessary concrete methods
*/
    void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }
    void makePlay(int player) {
        // Process a turn for the player
    }
    boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
    }
    void printWinner() {
        // Display the winning player
    }
    /* Specific declarations for the chess game. */

    // ...
}
```

Now we can extend this class in order to implement actual games

# Template Method: *Structure*



# Example: J2EE

- J2EE: A framework for “developing component-based multitier enterprise applications.”
- Example: class `TemplateFilter`
  - The abstract filter dictates the general steps that every filter must complete
  - The abstract filter leaves the specifics of how to complete that step to each filter subclass

# Example: Hook Methods in J2EE II

```
public abstract class TemplateFilter implements javax.servlet.Filter {
   ...
   public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
         // Common processing for all filters:
         doPreProcessing(request, response, chain);  // Hook 1
         doMainProcessing(request, response, chain); // Hook 2
         doPostProcessing(request, response, chain); // Hook 3
         ...
  }
  public void doPreProcessing(ServletRequest request,
     ServletResponse response, FilterChain chain) {}

  public void doPostProcessing(ServletRequest request,
     ServletResponse response, FilterChain chain) {}

  public abstract void doMainProcessing(ServletRequest
     request, ServletResponse response, FilterChain chain);
}
```

# Using J2EE

- How to use class `TemplateFilter`?
  - Extend and override the methods `doPreProcessing`, `doMainProcessing`, `doPostProcessing`

```
public class DebuggingFilter extends TemplateFilter {
  public void doMainProcessing(ServletRequest req,
    ServletResponse res, FilterChain chain) {
         System.out.println(“Filtering request:“ +
             req.asString());
   }
}
```

# The Template Method pattern: Applicability

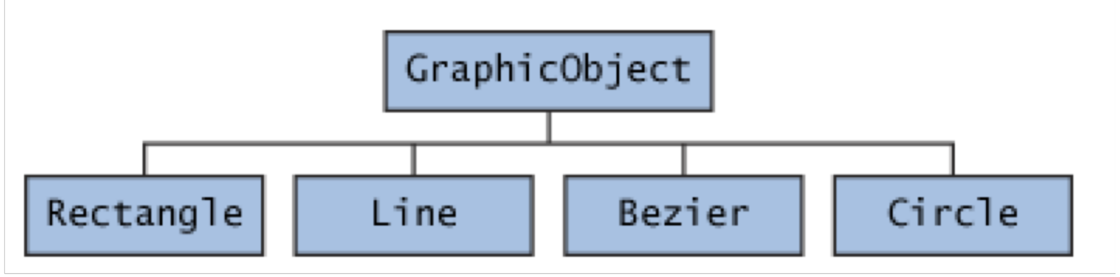
- Applicability: The Template Method pattern should be used
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "**refactoring to generalize**" as described by Opdyke and Johnson [OJ93]. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
  - to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

# The Template Method pattern: Participants

- Participants:
  - AbstractClass (e.g., TemplateFilter)
    - 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.
  - ConcreteClass (e.g., DebuggingFilter)
    - **implements the primitive operations** to carry out subclass-specific steps of the algorithm.

# Example Abstract Class

- In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and moveTo. Others require different implementations—for example, resize or draw. All GraphicObjects must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for **an abstract superclass**. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, GraphicObject, as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

# Example implementation

- First, you declare an abstract class, GraphicObject, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the **moveTo method**. GraphicObject also declares **abstract methods for methods, such as draw or resize**, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
            int x, y;
           ...
           void moveTo(int newX, int newY) {
            ...
  }
  abstract void draw();
  abstract void resize();
}
```

# Continued …

- Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the **draw** and **resize** methods:

```
class Circle extends GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
class Rectangle extends GraphicObject {
   void draw() {
      ...
   }
   void resize() {
      ...
   }
}
```

# Summary

- Looked at how patterns are used in software development in general
- Explored some common software development patterns
- For Analysis patterns see the Type diagrams lecture
- Design patterns
  - Template method pattern
    - Can be used in many different contexts
  - We will look at another design pattern (the Composite pattern) in the class in week 9

# Further reading

- Refactoring and Aggregation (1993), by Ralph E. Johnson , William F. Opdyke. In Object Technologies for Advanced Software, First JSSST International Symposium, volume 742 of Lecture Notes in Computer Science
- See Bennett Chapter 15 – Design Patterns