Design Patterns

Massimo Felici

School of informatics

Reuse in Software Engineering

- Software Engineering is concerned with processes, techniques and tools which enable us to build "good" systems
- Object-Orientation is a methodology, technique, process, suite of design and programming languages and tools with which we may build good systems
- Components are units of reuse and replacement

Examples of Types of Reuse

- **Application system reuse:** the whole of an application system may be reused by incorporating it without change into other systems
- **Component reuse:** components of an application ranging in size from sub-systems to single objects may be reused
- **Object and function reuse:** software components that implement a single function, such as a mathematical function or an object class, may be reused

Benefits of Software Reuse

- Increased dependability
- Reduced process risk
- Effective use of specialists
- Standards compliance
- Accelerated development

Slide 3: Benefits of Software Reuse

Increased dependability: Reused software, which has been tried and tested in working systems, should be more dependable than new software because its design and implementation faults have already been found and fixed.

Reduced process risk: The cost of existing software is already known, while the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.

Effective use of specialists: Instead doing the same work over and over, these application specialists can develop reusable software that encapsulates their knowledge.

Slide 3: Benefits of Software Reuse

Standards compliance: Some standards, such as user interface standards, can be implemented as a set of standard reusable components. For example, if menus in a user interface are implemented using reusable using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users are less likely to make mistakes when presented with a familiar interface.

Accelerated development: Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

Problems with Software Reuse

- Increased maintenance costs
- Lack of tool support
- Not-invented-here syndrome
- Creating and maintaining a component library
- Finding, understanding and adapting reusable components

Slide 4: Problems with Software Reuse

Increased maintenance costs: If the source code of a reused software system or component is not available the maintenance costs may be increased because the reused elements of the system may become increasingly incompatible with system changes.

Lack of tool support: CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library. The software process assumed by these tools may not take reuse into account.

Not-invented-here syndrome: Some software engineers prefer to rewrite components because they believe they can improve on them. This ids partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other peoples software.

Slide 4: Problems with Software Reuse

Creating and maintaining a component library: Populating a reusable component library and ensuring the software developers can use this library can be expensive. Our current techniques for classifying, cataloguing and retrieving software components are immature.

Finding, understanding and adapting reusable components: Software components have to be discovered in a library understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they will make include a component search as part of their normal development process.

Planning Reuse

- The development schedule for the software
- The expected software lifetime
- The background, skills and experience of the development team
- The criticality of the software and its non-functional requirements
- The application domain
- The platform on which the system will run

Slide 5: Approaches Supporting Software Reuse

- Design Patterns
- Component-based Development
- Application Frameworks
- Legacy system wrapping
- Service-oriented systems
- Application product lines
- COTS (Commercial-Off-The-Shelf) integration
- Configurable vertical applications
- Program libraries
- Program generators
- Aspect-oriented software development

Types of Reuse

- Reuse of Knowledge
  - Artifact reuse
  - Pattern reuse
- Reuse of Software
  - Code reuse
  - Inheritance reuse
  - Template reuse
  - Component reuse
  - Framework reuse

Artifact Reuse

- Reuse of use cases, standards, design guidelines, domain-specific knowledge
- Pluses: consistency between projects, reduced management burden, global comparators of quality and knowledge
- Minuses: overheads, constraints on innovation (coder versus manager)

Pattern Reuse

- A design pattern is a solution to a common problem in the design of computer systems
- Reuse of publicly documented approaches to solving problems (e.g., class diagrams)
- Plusses: long life-span, applicable beyond current programming languages, applicable beyond Object Orientation?
- Minuses: no immediate solution, no actual code, knowledge hard to capture/reuse.

Documenting Patterns

- Pattern Name
- Classification
- Intent
- Also Known As
- Motivation (Forces)
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Slide 9: Documenting Patterns

Pattern Name: a short name for the pattern (usally one or two words). This name should be indicative and explanatory of the pattern's purpose. It should also be unique within the application area.

Classification: each pattern is classified as **creational** (concerned with how objects are created), **structural** (concerned with putting objects together into a larger structure) or **behavioral** (concerned with collaborations between objects to achieve a particular goal).

Intent: a short description (one or two sentences) summarising what the pattern is for.

Also Known As: aliases for the pattern

Motivation (Forces): a description of the **design problem** that is addressed by the pattern

Applicability: areas where the pattern can be applied – the context for the pattern.

Structure: A graphical representation of the pattern – one or more class diagrams (and interaction diagrams) that illustrate how the pattern works.

Participants: a short description of the objects (classes) involved in the pattern and their responsibilities and roles.

Collaborations: a description of the collaborations between the participants – a description of how classes and objects used in the pattern interact with each other.

Consequences: a description of the benefits and shortcomings of the pattern – a description of the results, side effects, and trade offs caused by using the pattern.

Implementation: a description of an implementation of the pattern – advises on implementing the pattern.

Sample Code: an implementation of the pattern in a programming language.

Known Uses: examples of real usages of the pattern.

Related Patterns: other patterns that are similar to the one described – discussion of the differences between the patterns.

Slide 9: Documenting Patterns

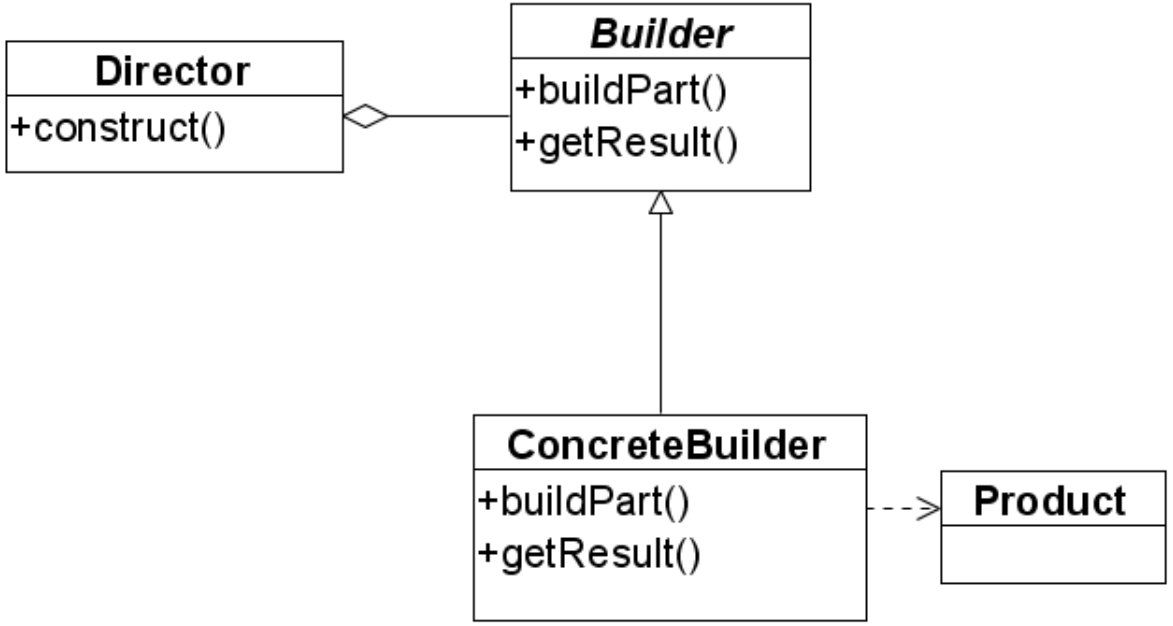
Suggested Readings

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

- M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Bushmann, P. Sommerlad. Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, 2006.

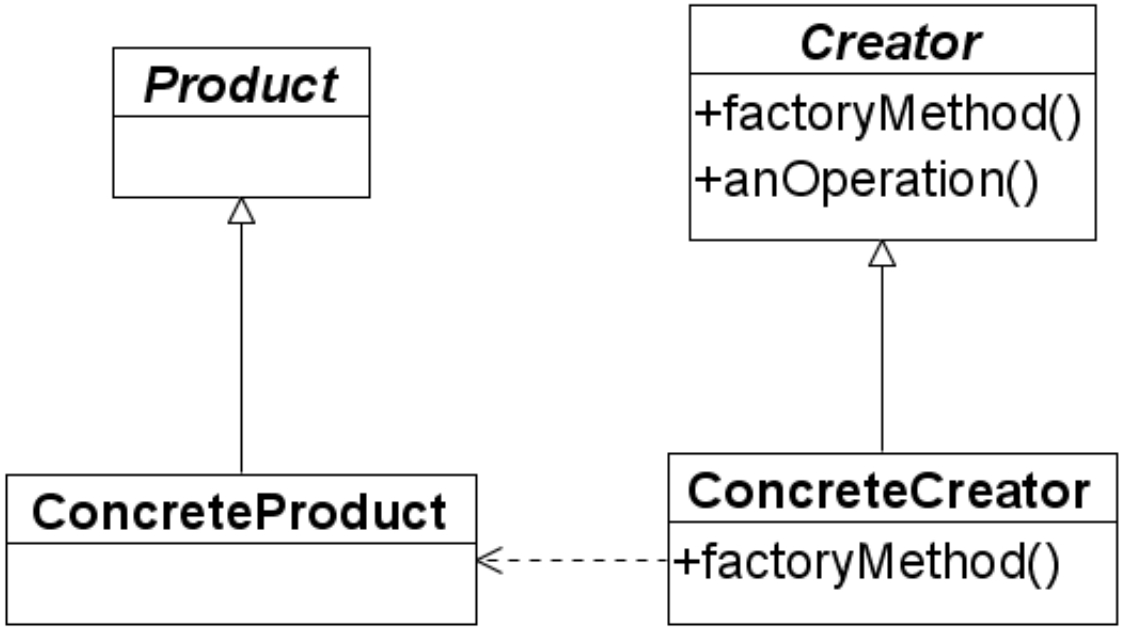
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations



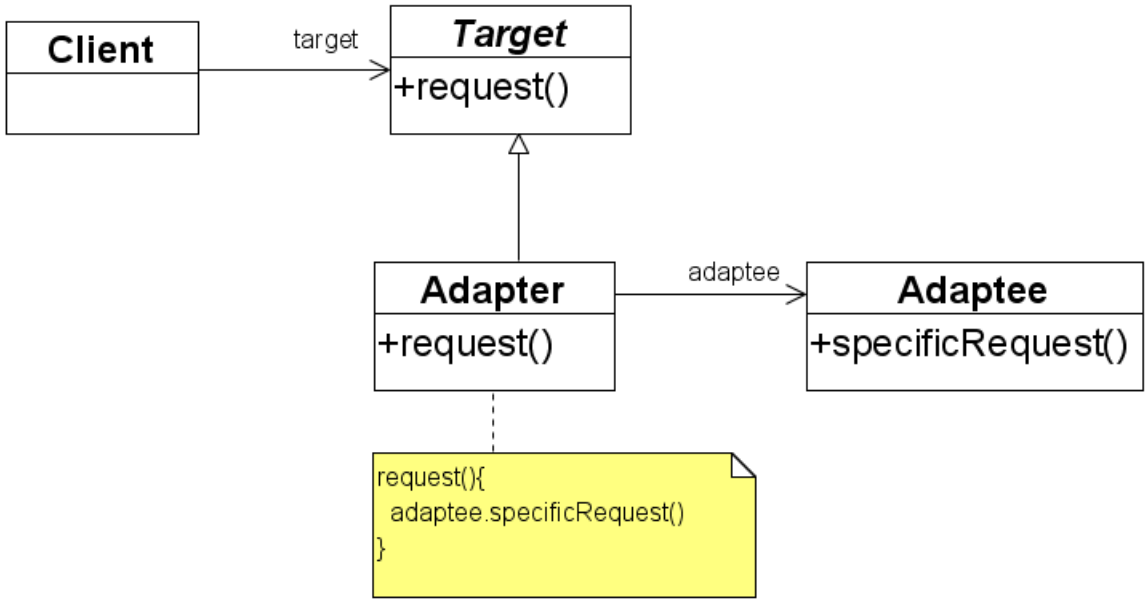
Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses



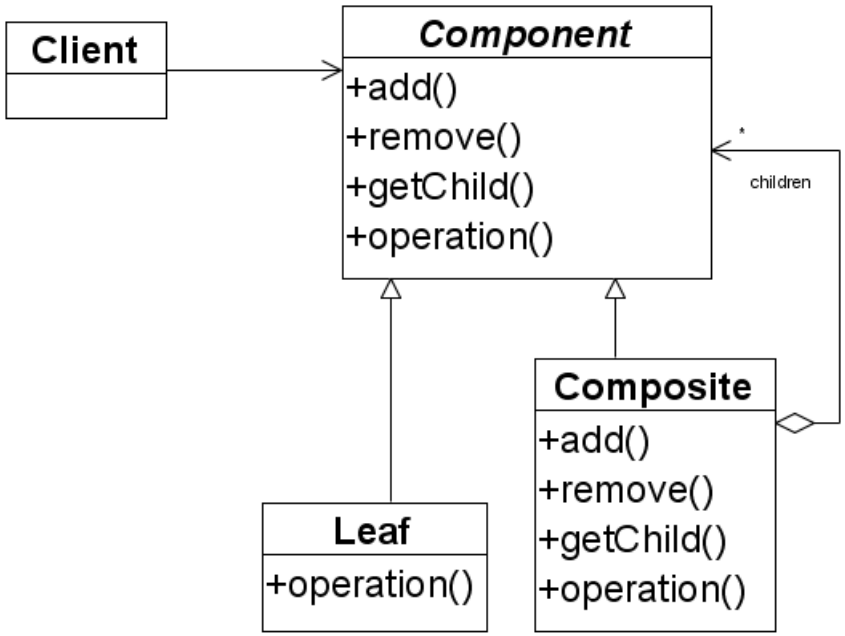
Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces



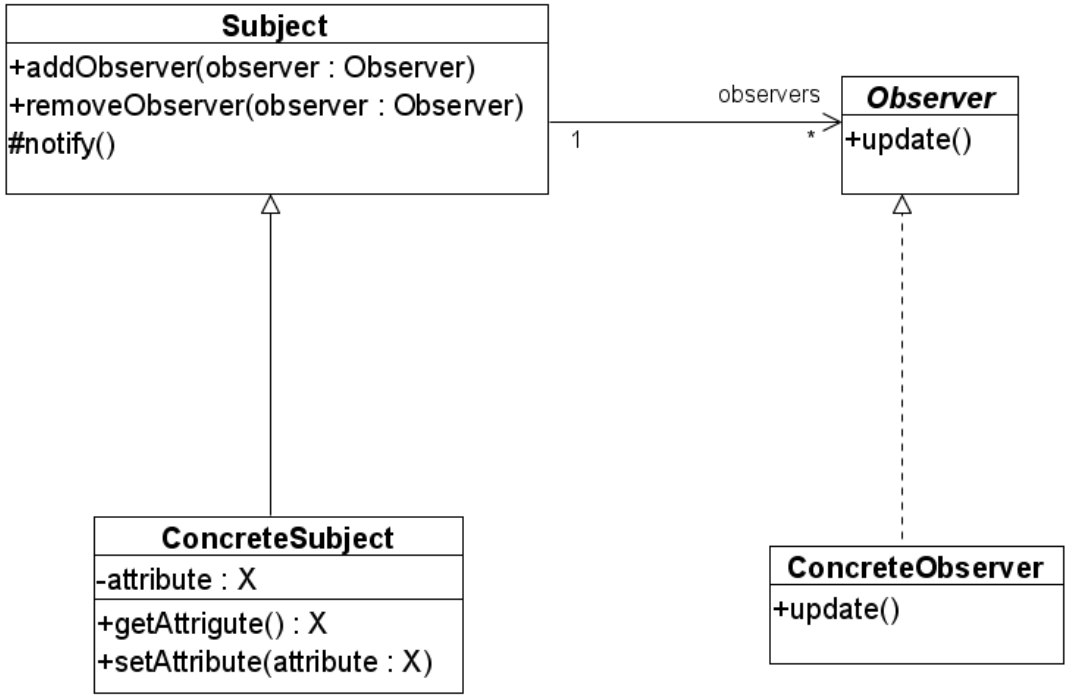
Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly



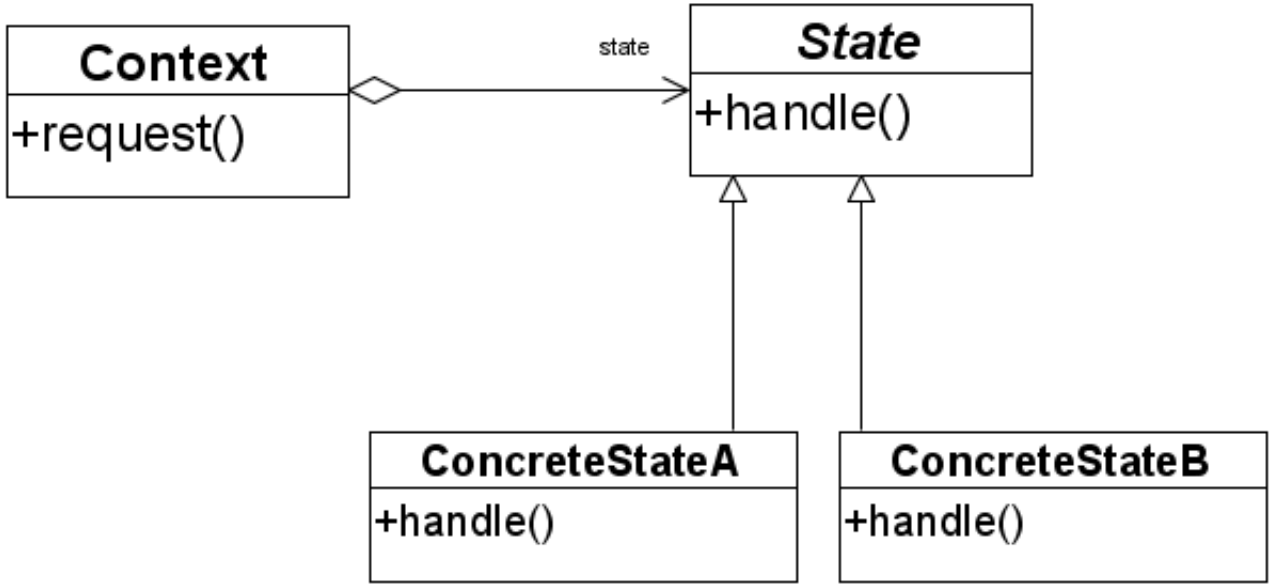
Observer

Define a one to many dependency between objects so that when one object changes state, all its dependents are notified automatically



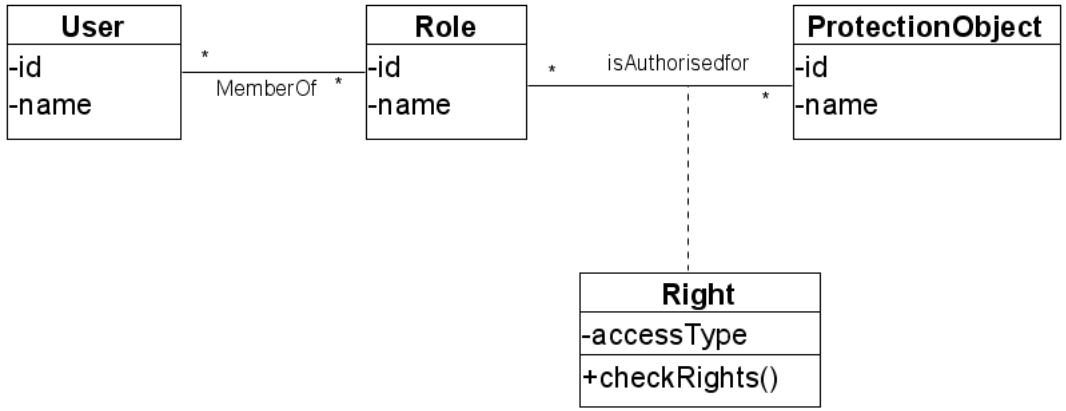
State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class



Role-Based Access Control

Assign rights based on the functions or tasks of people in an environment in which control of access to computing resources is required and where there is a large number of users, information types, or a large variety of resources – Describe how users can acquire rights based on their job functions or their task assigned



How to use a pattern

- Does a patter exist that address the considered problem?
- Does the patterns documentation suggest alternative solutions?
- Is there a simple solution?
- Is the context of the pattern consistent with the context of the problem?
- Are the results of using the pattern acceptable?
- Are there constraints?

Slide 17: Application of a Pattern

1. Read the pattern
2. Study its structure
3. Examine the sample code
4. Choose names for the patterns participants
5. Define the classes
6. Choose application-specific names for operation
7. Implement operations that perform the responsibilities and collaborations in the pattern

Code Reuse

- Reuse of (visible) source code - code reuse versus code salvage
- Pluses: reduces written code, reduces development and maintenance costs
- Minuses: can increase coupling, substantial initial investment

Inheritance

- Using inheritance to reuse code behaviour
- Pluses: takes advantage of existing behaviour, decrease development time and cost
- Minuses: can conflict with component reuse, can lead to fragile class hierarchy - difficult to maintain and enhance

Template Reuse

- Reuse of common data format/layout (e.g., document templates, web-page templates, etc.)
- Pluses: increase consistency and quality, decrease data entry time
- Minuses: needs to be simple, easy to use, consistent among groups

Component Reuse

- Analogy to electronic circuits: software "plug-ins"
- Reuse of prebuilt, fully encapsulated "components"; typically self-sufficient and provide only one concept (high cohesion)
- Pluses: greater scope for reuse, common platforms (e.g., JVM) more widespread, third party component development
- Minuses: development time, genericity, need large libraries to be useful

Framework Reuse

- Collection of basic functionality of common technical or business domain (generic "circuit boards") for components
- Pluses: can account for 80
- Minuses: substantial complexity, leading to long learning process, platform specific, framework compatibility issues leading to vendor specificity, implement easy 80

Reuse Pitfalls

- Underestimating the difficulty of reuse
- Having or setting unrealistic expectations
- Not investing in reuse
- Being too focused on code reuse
- Generalising after the fact
- Allowing too many connections

Slide 23: Reuse Pitfalls

- Underestimating the difficulty of reuse
  - Software must be more general
  - Similarities among projects often small
  - Much of what is reused is already provided by Operating Systems
  - Universe in constant flux (hardware, software, environment, requirements, expectations, etc.)
- Having or setting unrealistic expectations
  - Software is not built from basic blocks
  - Reuse domain may be unrealistic
  - Expectations for reuse outstrip skills of developers
- Not investing in reuse
  - Reuse costs: time and money in development, analysis, design, implementation, testing,...

Slide 23: Reuse Pitfalls

- Being too focused on code reuse
  - Focus on code reuse as end, not means
  - "Lines of code reused" metric meaningless
  - Design reuse often neglected in favour of code reuse
  - Too little abstraction at framework level
- Generalising after the fact
  - Design often migrate from general to specific during development
  - System not designed with reuse in mind (code reuse versus code salvage)
- Allowing too many connections
  - Coupling unavoidable, but must be very low to permit reuse
  - Circular dependencies also problematic - where to break the chains?

Difficulties with Component Development

- Economic
  - Small business do not have long term stability and freedom required
- Where is the third party component market?
  - Effort in (re)using components
  - Cross-platform and cross-vendor compatibility
  - Many common concepts, few common components
  - Some success: user interfaces, data management, thread management, data sharing between applications
  - Most successful: GUIs and data handling (e.g., Abstract Data Types)

Readings

Required Readings

- UML course textbook, Chapter 17 on Design Patterns
- T. Winn, P. Calder. Is This a Pattern?. IEEE Software 19(1):59-66, January/February, 2002.

Suggested Readings

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Bushmann, P. Sommerlad. Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, 2006.

Summary

- Many types of reuse (of both knowledge and software)
  - Each has pluses and minuses
- Component reuse is a form of software reuse
  - Encapsulation, high cohesion, specified interfaces explicit context dependencies
  - Component development requires significant time and effort, as does component reuse
  - Component reuse has been successful for interfaces and data handling
- Employing reuse requires management

