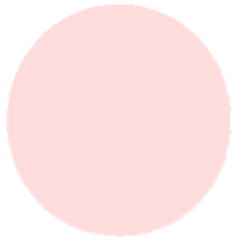


Design Patterns

Massimo Felici

IF 3.46   0131 650 5899

mfelici@inf.ed.ac.uk

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: 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.
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 reuse

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

© 2004-2008

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 people's software.

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: Key Factors

- 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

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

- Knowledge reuse
  - Artificial reuse
  - Pattern reuse
- Software reuse
  - Code reuse
  - Inheritance reuse
  - Template reuse
  - Components
  - Framework reuse

Reuse of Knowledge: 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)

Reuse of Knowledge: Patterns

- 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.

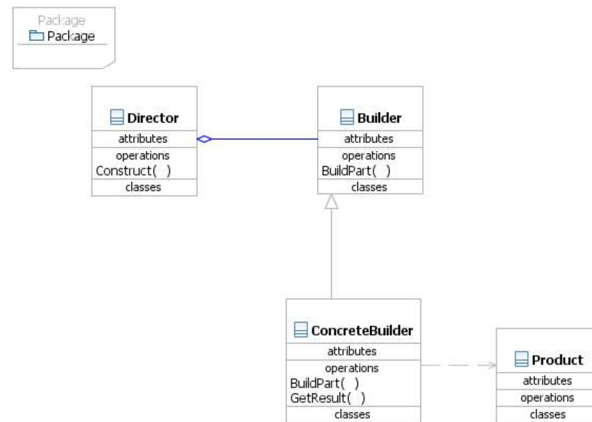
Documenting Patters

- Name
- Problem
- Context
- Forces
- Solution
- Sketch
- Resulting context
- Rationale

Classification of UML Patterns

- Creational
- Structural
- Behavioural

Example: Builder Pattern



© 2004-2008

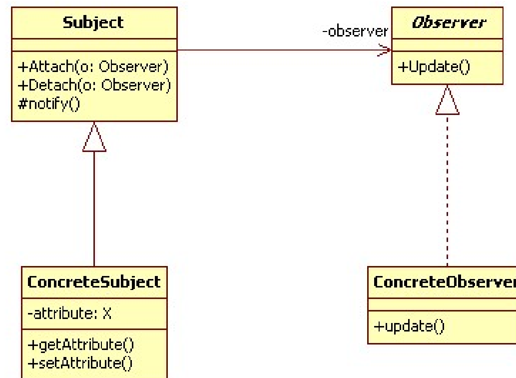
Director: The Director class is responsible for managing the correct sequence of object creation. It receives a Concrete Builder as a parameter and executes the necessary operations on it.

Builder: Abstract interface for creating objects (product).

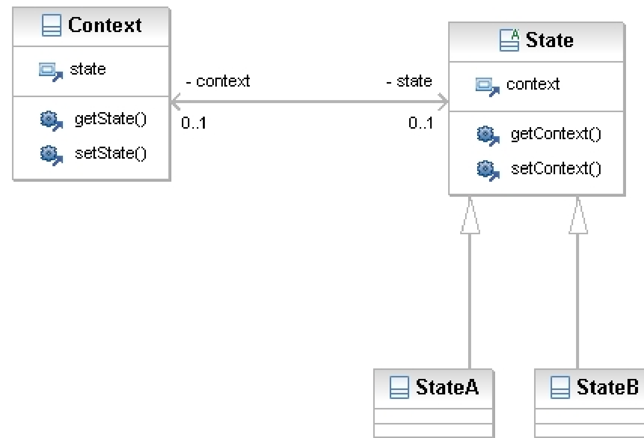
Concrete Builder: Provide implementation for Builder. Construct and assemble parts to build the objects.

Product: The complex object under construction.

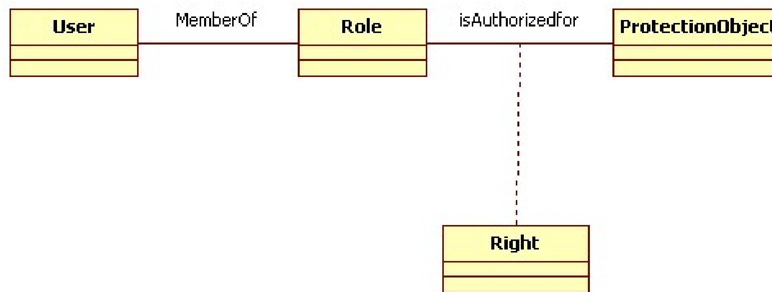
Example: Observer



Example: State



Example: Role-Based Access Control



How to use a pattern

- Does a patter exist that address the considered problem?
- Does the pattern's 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?

Application of a Pattern

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

Types of Software Reuse: 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

Types of Software Reuse: 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

Types of Software Reuse: 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

Types of Software Reuse: Component

- 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

Types of Software Reuse: Framework

- Collection of basic functionality of common technical or business domain (generic "circuit boards") for components
- **Pluses**: can account for 80% of code
- **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

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,…

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 (cf. 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

- **UML course textbook**
  - Chapter 17 on Design Patterns
- T. Winn, P. Calder, Is This a Pattern?. In IEEE Software, January/February 2002.

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