Introduction to Design Patterns

CS 251 Course Philosophy

*Good design and programming is not learned by generalities, but by seeing how significant programs can be made clean, easy to read, easy to maintain and modify, human-engineered, efficient, and reliable, by the application of good design and programming practices. Careful study and imitation of good designs and programs significantly improves development skills.*

- *Kernighan and Plauger*
Goals

Show *by example* how patterns and frameworks can help to

- Codify good Object Oriented (OO) software design and implementation practices
  - Distill and generalize experience
  - Aid novices and experts alike
- Give design structures explicit names
  - Common vocabulary
  - Reduced complexity
  - Greater expressivity
- Capture/preserve design and implementation knowledge
  - Articulate key decisions succinctly
  - Improve documentation
- Facilitate restructuring/refactoring
  - Patterns and frameworks are interrelated
  - Enhance flexibility, reuse, and productivity

```
class Reactor {
    public:
        /// Singleton access point.
        static Reactor *instance (void);
        ///< Run event loop.
        void run_event_loop (void);
        ///< End event loop.
        void end_event_loop (void);
        ///< Register @a event_handler
        void register_input_handler (Event_Handler *eh);
        ///< Remove @a event_handler
        void remove_input_handler (Event_Handler *eh);
    }
```

Design Patterns: Motivation and Concepts

- Object oriented design methods emphasize design notations
- Fine for specification and documentation
Design Patterns: Motivation and Concepts

• Object oriented design (OOD) methods emphasize design notations
  • Fine for specification and documentation
  • But OOD is more than just drawing diagrams
  • Good draftsmen are not necessarily good architects!

• Good object oriented designers rely on lots of experience
  • At least as important as syntax
• Most powerful reuse combines design and code reuse
  • Patterns: Match problem to design experience
  • Frameworks: Reify patterns within a domain context
Recurring Design Structures

Well-designed object oriented systems exhibit recurring structures that promote
- Abstraction
- Flexibility
- Modularity
- Elegance

Therein lies valuable design knowledge

Problem: capturing, communicating, applying, and preserving this knowledge without undue time, effort, and risk

A Pattern...

- Abstracts and names a recurring design structure
- Comprises class and/or object
  - Dependencies
  - Structures
  - Interactions
  - Conventions
- Specifies the design structure explicitly
- Is distilled from actual design experience

Presents solution(s) to common (software) problem(s) arising within a context
Four Basic Parts of a Pattern

1. Name
2. Problem (including “forces” and “applicability”)
3. Solution (both visual and textual descriptions)
4. Consequences and trade-offs of applying the pattern

Key characteristics of patterns include:
• Language- and implementation-independent
• “Micro-architecture,” i.e., “society of objects”
• Adjunct to existing methodologies (RUP, Fusion, etc.)

Design Space for Gang of Four (GoF) Patterns

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Factory Method ✓</td>
<td>Adapter (class) ✓</td>
<td>Interpreter ✓</td>
</tr>
<tr>
<td></td>
<td>Builder ✓</td>
<td>Bridge ✓</td>
<td>Chain of Responsibility ✓</td>
</tr>
<tr>
<td></td>
<td>Prototype ✓</td>
<td>Composite ✓</td>
<td>Iterator ✓</td>
</tr>
<tr>
<td></td>
<td>Singleton ✓</td>
<td>Decorator ✓</td>
<td>Mediator ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flyweight ✓</td>
<td>Memento ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Facade ✓</td>
<td>Observer ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy ✓</td>
<td>State ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy ✓</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor ✓</td>
</tr>
</tbody>
</table>

**Scope**: domain over which a pattern applies

**Purpose**: reflects what a pattern does
GoF Pattern Template (1st half)

Intent
Short description of the pattern and its purpose

Also Known As
Any aliases this pattern is known by

Motivation
Motivating scenario demonstrating pattern’s use

Applicability
Circumstances in which pattern applies

Structure
Graphical representation of pattern using modified UML notation

Participants
Participating classes and/or objects and their responsibilities

GoF Pattern Template (2nd half)

Collaborations
How participants cooperate to carry out their responsibilities

Consequences
The results of application, benefits, and liabilities

Implementation
Pitfalls, hints, techniques, plus language-dependent issues

Sample Code
Sample implementations in C++, Java, C#, Python, Smalltalk, C, etc.

Known Uses
Examples drawn from existing systems

Related Patterns
Discussion of other patterns that relate to this one
Overview of Pattern Sequences and Languages

**Motivation**
- Individual patterns and pattern catalogs are insufficient
- Software modeling methods and tools largely just illustrate what/how – not why – systems are designed

**Benefits of Pattern Sequences & Languages**
- Define vocabulary for talking about software development problems
- Provide a process for the orderly resolution of these problems, e.g.:
  - What are key problems to be resolved and in what order
  - What alternatives exist for resolving a given problem
  - How should mutual dependencies between the problems be handled
  - How to resolve each individual problem most effectively in its context
- Help to generate and reuse software architectures
Benefits and Limitations of Patterns

**Benefits**

- Design reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, and team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

**Limitations**

- Require significant tedious and error-prone human effort to handcraft pattern implementations design reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves important (implementation) details unresolved

Addressing the limitations of patterns requires more than just design reuse

---

Overview of Frameworks

- Frameworks provide a set of cooperating classes that make up a reusable design for a particular class of software, such as graphical user interfaces.
  - Dictates the architecture of the application, including overall structure, partitioning into classes and objects, key responsibilities, collaboration between objects, etc.
  - Captures the common design decisions for the application domain.
  - Emphasizes design reuse over code reuse.
Overview of Frameworks

- Frameworks exhibit "inversion of control" at runtime via callbacks
- Frameworks provide integrated domain-specific structures & functionality
- Frameworks are "semi-complete" applications

Motivation for Frameworks

Legacy embedded systems have historically been:
- Stovepiped
- Proprietary
- Brittle and non-adaptive
- Expensive
- Vulnerable

Consequence: Small HW/SW changes have big (negative) impact on system QoS and maintenance
Motivation for Frameworks

- **Frameworks** factor out many reusable general-purpose and domain-specific services from traditional distributed real-time and embedded application responsibility
- Essential for **product-line architectures (PLAs)**
- Product-lines and frameworks offer many configuration opportunities
  - e.g., component distribution/deployment, OS, protocols, algorithms, etc.

Categories of OO Frameworks

- **White-box frameworks** are reused by subclassing, which usually requires understanding the implementation of the framework to some degree
- **Black-box framework** is reused by parameterizing and assembling framework objects, thereby hiding their implementation from users
- Each category of OO framework uses different sets of patterns, e.g.:
  - **White-box frameworks** rely heavily on inheritance-based patterns, such as Template Method & State
  - **Black-box frameworks** rely heavily on object composition patterns, such as Strategy and Decorator

Many frameworks fall in between white-box and black-box categories
Commonality and Variability Analysis in Frameworks

- Framework characteristics are captured via **Scope, Commonalities, and Variabilities (SCV)** analysis
- This process can be applied to identify commonalities and variabilities in a domain to guide development of a framework

Applying SCV to avionics mission computing
- Scope defines the domain and context of the framework
- Component architecture, object-oriented application frameworks, and associated components, e.g., GPS, Airframe, and Display

Applying SCV to an Avionics Framework

- **Commonalities** describe the attributes that are common across all members of the framework
- Common object-oriented frameworks and set of component types
  - e.g., GPS, Airframe, Navigation, and Display components
- Common middleware infrastructure
  - e.g., Real-time CORBA and a variant of Lightweight CORBA Component Model (CCM) called Prism
Applying SCV to an Avionics Framework

- **Variabilities** describe the attributes unique to the different members of the framework
- **Product-dependent component implementations** (GPS/INS)
- **Product-dependent component connections**
- **Product-dependent component assemblies** (e.g., different weapons systems for different customers/countries)
- **Different hardware, OS, and network/bus configurations**

Comparing Reuse Techniques

**Class Library (and STL) Architecture**
- A **class** is an implementation unit in an OO programming language, i.e., a reusable type that often implements **patterns**
- Classes in class libraries are typically **passive**

**Framework Architecture**
- A **framework** is an integrated set of classes that collaborate to form a reusable architecture for a family of applications
- Frameworks implement **pattern languages**

**Component and Service-Oriented Architecture**
- A **component** is an encapsulation unit with one or more interfaces that provide clients with access to its services
- Components can be deployed and configured via **assemblies**
Taxonomy of Reuse Techniques

<table>
<thead>
<tr>
<th>Class Libraries</th>
<th>Frameworks</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-level</td>
<td>Meso-level</td>
<td>Macro-level</td>
</tr>
<tr>
<td>Stand-alone</td>
<td>&quot;Semi-complete&quot;</td>
<td>Stand-alone</td>
</tr>
<tr>
<td>language entities</td>
<td>applications</td>
<td>composition entities</td>
</tr>
<tr>
<td>Domain-independent</td>
<td>Domain-specific</td>
<td>Domain-specific or Domain-independent</td>
</tr>
<tr>
<td>Borrow caller’s thread</td>
<td>Inversion of control</td>
<td>Borrow caller’s thread</td>
</tr>
</tbody>
</table>

Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation and deployment of software
Benefits of Frameworks

- **Design reuse**
  - e.g., by guiding application developers through the steps necessary to ensure successful creation and deployment of software

- **Implementation reuse**
  - e.g., by amortizing software lifecycle costs and leveraging previous development and optimization efforts

- **Validation reuse**
  - e.g., by amortizing the efforts of validating application- and platform-independent portions of software, thereby enhancing software reliability and scalability
Limitations of Frameworks

- Frameworks are powerful, but can be hard to use effectively (and even harder to create) for many application developers.
- Commonality and variability analysis requires significant domain knowledge and OO design/implementation expertise.
- Significant time required to evaluate applicability and quality of a framework for a particular domain.
- Debugging is tricky due to inversion of control.
- Validation and verification is tricky due to “late binding.”
- May incur performance degradations due to extra (unnecessary) levels of indirection.

Many frameworks limitations can be addressed with knowledge of patterns!

Using Frameworks Effectively

Observations

- Since frameworks are powerful—but hard to develop and use effectively by application developers—it is often better to use and customize COTS frameworks than to develop in-house frameworks.
- Classes/components/services are easier for application developers to use, but are not as powerful or flexible as frameworks.

Successful projects are therefore often organized using the “funnel” model.
Stages of Pattern and Framework Awareness

benefit

familiarity

understanding

initiation

consternation

ignorance

31