Software Engineering
G22.2440-001

Session 8 – Sub-Topic 2
Guidelines for Architecting Software

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Part I

Role of the Architect
Section Objectives

• Examine Tactics for architecting for non-functional requirements

Responsibilities

• Providing decisive technical leadership
• Providing a clearly articulated technical vision
• Building a cohesive and motivated team
• Providing governance to make sure the architecture is being implemented correctly
• Interact with stakeholders to ensure that their requirements are clearly stated and met
• Communicating technical issues to management
• Ensure that the architecture addresses operational, deployment and maintenance issues
Responsibilities (2)

- Evangelize! Champion your architecture!
- Making justifiable technical tradeoffs
- Resolution of technical problems
- Maintain morale through clear technical direction
- Provide input for tool and environment selection
- Manage risk identification and mitigation standards
- Plan for evolutionary changes, such as legacy migration

Technical Leadership

- Vision, Integrity and Decisiveness
  - Vision: establishing a concrete goal for the team to move towards - how else can you justify technical decisions you make?
  - Integrity: motivating the team to do what’s best for the project not the easiest for them
  - Decisive: making solid technical decisions quickly
Vision

• Clearly articulate the project vision
• Motivate the team towards realizing that vision
• Provides a basis for articulating and justifying various technical trade-offs
• A clear vision provides an identifiable path of success to realizing the project goals

Integrity

• Technical solutions ideally should be as a result of a technical decision making process
• Decisions made on any other basis (e.g. political, vendor) represent de-motivating factors and compromise the overall vision
• “we had to use [censored], it was chosen by [censored] and really sucks” - [censored] 2002
Being Decisive

• Demonstrating technical decisiveness is a key factor in establishing confidence in your leadership
• Making a decision, *any decision*, is far better than avoiding the issue

Team Building

• The architect is the most visible technical member of the team
• Key role - continually articulate the *strategic* technical vision to development staff and management
• This facilitates a greater acceptance of day to day *tactical* decisions
Focus on Technology

- Team members respond better to technical drivers than interpersonal ones
- Interpersonal drivers often call underlying motives into question
- Solving technical issues keep the project focused on achieving its goals

Trust

- Trust in the capabilities of your team
- This involves a degree of personal and professional risk, but comes with the responsibility you are undertaking
- The project can only succeed if a majority of the team members pull their weight, demonstrating professionalism and skill
Avoid Micromanagement

✓ A classic mistake, and originates from a lack of trust in team capabilities
✓ Two possible outcomes:
  1. The development team becomes resentful that the architect doesn’t trust them to do their jobs in a professional manner
  2. The development team hands over a large chunk of implementation responsibility to the architect, who fails to meet productivity expectations and confidence in him/her plummets

Governance

• The architect is the main enforcer of architectural governance
• This is conformance to the standards set by the architecture in terms of structure and style
Communicating with Management

• Architects often required to produce project updates for senior management
  – Tendency to devolve into too much technical detail
  – Keeps reports factual but short. Summaries are far more digestible than extensive technical diatribe!

Stakeholder Management

• A “stakeholder” represents an individual/organization/customer who has a “stake” in the project outcome
  – Typically this is a financial stake
  – But encompasses both political issues and the broader user community
Stakeholder Management (2)

• Initially the architect will typically communicate the proposed architecture(s) to the stakeholder for analysis and feedback
  – This may include a number of alternatives
  – Will include both system architecture and tools, processes and environments
  – Designed to establish confidence that:
    • The emerging requirements will be met by the architecture
    • The requirements have been understood and communicated directly to the architect

Stakeholder Management (3)

• Architect will continue to work with the stakeholders, communicating technical progress, emerging risks, and other developments as the project proceeds
Architects and Project Managers

• Important to define the key differentiators between PM’s and Architects
• Architect responsibilities
  – Technical planning, staffing, tracking
  – Maintaining project quality and technical vision
• PM responsibilities
  – Scheduling, Budgets, Resource Allocation, Hiring, Customer Management…

Interaction During Planning

• Architects provide a PM with the following capabilities:
  – Do benefit/risk analysis of products and environments
  – Provide the work breakdown for H/W and S/W tasks
  – Define tasks for the schedule to include quality management, e.g. walkthroughs, reviews…
  – Assist the PM in team building efforts
Interaction During Analysis

• Architect communicates the following to the PM
  – Chosen architectural alternatives and tradeoffs
  – How functional and non-functional requirements will be addressed
  – Implementation alternatives

Interaction During Implementation

• Communicates ongoing technical risks to the PM for evaluation and mitigation planning
• Works with the PM to monitor the quality of project deliverables, such as through structured code reviews
Mediation

- The architect often performs a key mediating role between development staff, managers and stakeholders
- Developers are your primary resource, they should always be your first concern
  - Trust their instincts
  - Make sure they receive the training, tools and software to get their job done effectively
  - Make sure they receive appropriate recognition for their efforts

Technical Tradeoffs

- Required to make decisions regarding competing architectural capabilities:
  - Examples
    - Performance versus Maintainability
    - Security versus Performance
    - Everything versus Cost!
Technical Tradeoff Example

• Hard Real-Time System Development
  – System must deliver a result by a specified deadline or be incorrect
  – System partitions work into one or more threads, each with a separate deadline
  – However, performance estimates indicate that the scheduling overhead required to execute these processes will cause the system to meet its deadlines.

Technical Tradeoff Example (2)

• The solution - an architectural tradeoff
  – Combine all the threads into a single unit of execution (this is typically called a cyclic executive), essentially one giant loop!
  – Now meets performance requirements, but has disadvantageous consequences for understandability and maintainability
  – A classic tradeoff, the structure and clarity of the architecture is compromised for performance needs
Input to Tool/Environment Selection

• The architect will have a critical role in the selection of development tools and environments
  – E.g. server, OS, IDEs, Modeling tools, source control, Business Rules Engines…
• Will be required to undertake research to analyze a given vendor solution space
  – May be required to justify cost benefits

Managing Technical Risk

• Will work with the PM to eliminate, mitigate or remove risk
  – “a variable that can have a value that endangers or eliminates project success criteria”
• Typically the architect will provide an early risk assessment
  – Measured by severity and probability
Risk Management Strategies

• Risk Avoidance  
  – Avoid the risk altogether  
  – Also known as “dodging the bullet”

• Risk Transfer  
  – Give the risk to someone / something else  
  – Also known as “sloping shoulders”

• Risk Acceptance  
  – Decide to live with the risk  
  – Also known as “deal” or “biting the bullet”  
  – Requires either a Mitigation Strategy or Planning for Contingency

Part II

CRC Cards
CRC Overview

The CRC card technique is an informal technique for object oriented analysis and design.

It was invented by Ward Cunningham and Kent Beck in 1989 and then popularized by Rebecca Wirfs-Brock.

It is an excellent technique for visualizing the structure and behaviour of OO systems.

It is an excellent vehicle for learning to think in an OO manner.

CRC Cards

CRC stands for Class, Responsibility, Collaborator

A CRC card is (usually) a standard index card

The Class name

The Classes responsibilities

The Classes collaborators
What is a ...

**Class**  
An abstraction of something from the problem domain.

**Responsibility**  
A task that can be ascribed to a particular class.

**Collaborator**  
Another class that you require to talk to in order to carry out your own responsibilities.

Characterizing the Responsibility Approach

A **data-driven** approach describes a horse in terms of its parts: head, tail, body, leg(4).

A **procedural** approach describes a horse in terms of operations it can perform: walk, trot, bite, eat, neigh.

A **responsibility** driven approach describes a horse in terms of its responsibilities: communicate, carry things, maintain its living systems.
CRC Card Syntax Example

Book - the set of objects that represent books that may be borrowed from the library

Superclass: Lendable

Subclass:
- know borrower
- know due date
- check out

Date
Fine

Organizing a CRC Card Session

The number of people is important, too few and it becomes difficult to mentally change gear, too many and you are not all involved. 5 is the ideal number.

Try to avoid having an excessively dominant participant - no bosses either.

It is meant to be fun and it could be noisy so find a place away from the rest of the world.

From the system requirements decide on a simple set of scenarios that you want to try out.
Conducting a CRC Session

Start the session by brainstorming possible classes from the domain. Each person should take the responsibility for being a class.

Select a scenario. Identify and allocate new classes and responsibilities as the scenario unfolds.

Each person has to focus on their own class. *I am a bank account, I know my own balance and I have to watch out for going overdrawn.*

The list of classes will grow and then shrink as the group filters out the good ones.

The interaction should be recorded when the group are happy with it.

CRC Good Practice

- Start with the simplest scenarios.
- Take the time to select meaningful class names.
- Take the time to write a description of the class.
- If in doubt, act it out!
- Layout the cards on the table to get an intuitive feel for system structure.
- Be prepared to be flexible.
Part III

Visual Modeling with UML

Section Objectives

• Examine the advantages of visual modeling
• Introduce the following aspects of the UML
  – Use Cases
  – Class Diagrams
  – Behavior Diagrams
  – Implementation Diagrams
  – Object Constraint Language
What is Visual Modeling?

- The use of “semantically rich” textual and graphical notations to model software systems
- Provide a satisfactory degree of formality and rigor at a higher level of abstraction than implementation
- Represents a standard basis of design communication and review
- Tries to be as unambiguous as possible to facilitate a clean transition to implementation

Modeling Advantages

- Facilitate understanding of complex systems
- Explore and compare design alternatives at low cost
- Provide a formal and unambiguous basis for code
- Provide a common basis for capturing and communication requirements
- Communicating decisions unambiguously
Managing Complexity

- Visual Modeling facilitates the derivation of “views” that isolate and emphasize system features and de-emphasize others
- Allows the designer to focus on the “big picture”
  - Use cases to capture requirements
  - Class diagrams to capture static partitioning into classes, relationships, attributes and operations
  - State diagrams to represent “interesting” dynamic behavior

Exploring Design Alternatives

- Modeling supports a degree of formality, diminishes ambiguity and provides a standard basis for communications
- Provides a solid basis for the comparison of static and dynamic design alternatives
- …AND IT IS CHEAPER THAN WRITING CODE!
A Basis for Implementation

- RUP uses UML as a basis for Object-Oriented System Implementation
- Tools such as Rational XDE permit design mapping activities:
  - Forward Engineering: uses the design model to facilitate the generation of sophisticated code skeletons
  - Reverse Engineering: uses the code to extract and generate a UML-compliant design model from the code base
  - Round Trip Engineering: the process of forward and reverse engineering used to keep the design and code artifacts synchronized

Requirements Capture

- Provides a basis for capturing requirements in a “stakeholder friendly” way to facilitate review by technical and non-technical individuals
- Focuses on “what” the system must do, can de-emphasize or even remove altogether “how” it will actually be done
Communication

- "It serves as a language for communicating decisions that are not obvious or cannot be inferred from the code itself."
- "It provides semantics that are rich enough to capture all important strategic and tactical decisions."
- "It offers a form concrete enough for humans to reason and for tools to manipulate." [Booch 95]

What is UML?

- “The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.”

- UML is not a programming language, a tool specification or a description of a software process.
UML: Specific Aims

- Programming language independent
- Extensible around a well defined core
- Have a formal basis
- It will be capable of supporting high level concepts such as patterns and frameworks
- Recursively defined with a meta-model (a what???)
- Enable tool development

Coverage

- Use case diagrams
- Class diagrams
- Behaviour diagrams
  - Sequence
  - Collaboration
  - Statechart
  - Activity
- Implementation diagrams
  - Component
  - Deployment
- Object constraint language
Use Cases

Order System
- Check Status
- Make Order
- Fulfill Orders
- Establish Credit

Customer
- Service Representative
- Shipping Clerk
- Credit Supervisor

What is a Class?

- “A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.”
- We model domains & build OO systems by declaring classes that define their instances
Classes - Purpose

• The first purpose deals with structural definition. A class provides a definition of the structure of instances of that class. The class defines the attributes, operations and relationships of all objects belonging to this class.

• The second purpose of a class is the creation of new objects. This is achieved through constructor operations.

• Finally a class may administer the set of all instances of that class.

Notation for Classes

Account

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Attribute Compartment</th>
<th>Operation Compartment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>number: int</td>
<td>addInterest()</td>
</tr>
<tr>
<td></td>
<td>balance: Money</td>
<td>setOverdraftLevel()</td>
</tr>
<tr>
<td></td>
<td>accountHolder: String</td>
<td></td>
</tr>
<tr>
<td></td>
<td>interestRate: int</td>
<td></td>
</tr>
</tbody>
</table>

Account

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
</tr>
<tr>
<td>balance</td>
</tr>
<tr>
<td>accountHolder</td>
</tr>
<tr>
<td>interestRate</td>
</tr>
</tbody>
</table>

Account

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>addInterest()</td>
</tr>
<tr>
<td>setOverdraftLevel()</td>
</tr>
</tbody>
</table>
**Class Details**

### Account

- `+ number: int`
- `# balance: Money`
- `# accountHolder: String`
- `- interestRate: int = 7`
- `- lastAccountId: String`
- `- addInterest(d: Date)`
- `+ update()`
- `# setOverdraftLevel()`
- `+ getBalance(): Money`

**Visibility**

- Initial value
- Parameter
- Return type

**Class scope**

- attendee: Person

**Notation for Objects**

- `attendee: Person`
- `: Frame`
- `: Card`
- `multiObject`
- `max: Person`
- `shoeSize = 11`
- `Object attributes`
Dependency

Account

- balance: Money
- accountHolder: String
- interestRate: int

- addInterest()
- setOverdraftLevel()
- deposit(cash: Money)

Money

Static Diagrams

Context

- ContextInterface()

Strategy

- Data
- AlgorithmInterface()

ConcreteStrategyA

- AlgorithmInterface()

ConcreteStrategyB

- AlgorithmInterface()

ConcreteStrategyC

- AlgorithmInterface()
Associations are implicitly bidirectional

Aggregation

- When we want to say that one thing is part of another
Aggregation and Composition

- Polygon
- Graphics Bundle
- Point

Generalization/Specialization

- Generalization: to abstract class Animal from Dog and Cat
- Specialization: to extend class Animal by adding Dog and Cat
Polymorphism

- Different classes provide the same operation
- External interface is the same
- Implementation is different
- ‘Client’ object issues request
- ‘Server’ object handles request

Sing!

- Type-checking is automatic
- Each class is simplified
- Easy to add new types

Behavior Diagrams

- Statechart diagram
- Interaction diagrams
  - Sequence diagrams
  - Collaboration diagrams
The notation used is David Harel’s Statechart notation. This allows for nested states and concurrent states.

Concurrent States

The diagram shows a statechart for an overhead projector with states for different positions and light states.
Sequence Diagrams

Sequence Diagrams and Use Cases
Sequence Diagram Shapes

- **Fork - centralized**
  - operations can change order
  - new operations may be added

- **Stair - decentralized**
  - Operations have strong connections
  - performed in same order
  - behaviour is encapsulated

Collaboration Diagrams

```
Order EntryWindow
  1: prepare()

Order
  2: prepare()
  3: check()
  4: \{check="true"\} remove()
  5: needToReorder()

redLine :Order Line
  6: new()

redStock :Stock Item
  7: \{check="true"\} new()

Delivery Item

RedOrder Item
  6: new()
```
Activity Diagrams

Implementation Diagrams

- Component diagrams
  - Shows the dependencies among software components
- Deployment diagrams
  - Shows the configuration of run-time processing elements
Implementation Diagrams

• There are 2 types of implementation diagrams, both concerned with the physical details of system organization.
• Component diagrams which show the relationships between software artifacts.
• Deployment diagrams which show the relationships between hardware and software artifacts.

Describing System Architectures

Design view  Implementation view
Use Case View
Process view  Deployment view
Component Diagrams

“A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.”  

“a physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of interfaces.”

Components

Like all classifiers in UML, components have types and instances.
Components and Code

Component Stereotypes

«executable» Specifies a component that may be executed on a node

«library» Specifies a static or dynamic object library

«table» Specifies a component that represents a database table

«file» Specifies a component that represents a document containing source code or data

«document» Specifies a component that represents a document
Nodes

“A node is a physical element that exists at runtime and represents a computational resource, generally having at least some memory and, often, processing capability.”

Connecting Nodes

- **t: Terminal**
  - Deploys user.exe
  - «ethernet»

- **s: Server**
  - Deploys dbadmin.exe

- **c: Console**
  - Deploys admin.exe
  - «RS-232»
Stereotyped Nodes

- **t:** Terminal
- **s:** Server
- **c:** Console

- Deploys `user.exe`
- «ethernet» connection
- «RS-232» connection

Nodes with Components

- **AdminServer:** HostMachine
- **Joe’sMachine:** PC

- **:Scheduler**
  - «database»
  - «meetings»

- **:Planner**

Components participate in the execution of a system; nodes execute components.
Object Constraint Language

- A formal language
- Developed within IBM and based on the Syntropy method
- Has simple constructs including arithmetic, boolean and predicate logic operators.
- Has built in data types for collections (sets, lists bags etc..)

OCL Examples

- An interface can only contain operations
  
  self.allFeatures->forall (f | foclIsKindOf(Operation))

- All features defined in an interface are public
  
  self.allFeatures->forall (f | f.visibility = #public)
Completeness

- There is an implicit assumption of a complete underlying model.
- Many uses of the notation will be, by definition, incomplete.
- The only model that could be “complete” is the implementation model.
- Defining a process is all about controlling the incompleteness.

Part IV

Architectural Evaluation
Learning Objectives

• Review best practices for process and architectural design
• Provide a detailed overview of heuristic assessment
• Provide an introduction to the application of metrics

Evaluation

• This presentation looks at the issues involved in the evaluation of architectures constructed using object-oriented technology
• There are 2 main approaches to OO design evaluation.
  • Heuristic based design assessment (informal)
  • Metric based design assessment (formal)
### What Makes an Architecture Good?

<table>
<thead>
<tr>
<th>Quality</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>Testability</td>
</tr>
<tr>
<td>Reliability</td>
<td>Flexibility</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Portability</td>
</tr>
<tr>
<td>Integrity</td>
<td>Reusability</td>
</tr>
<tr>
<td>Usability</td>
<td>Interoperability</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Understandability</td>
</tr>
</tbody>
</table>

### What Makes OO Different?

**System structure**

The components are subsystems, classes (objects), methods.

**Inheritance**

Superclass/subclass connections are absent from structured programs.

**Message Passing**

System behaviour is an emergent property of message passing.
A Sample Heuristic

- Systems of around 100 classes should have a depth of inheritance hierarchy of approximately 7±2 levels.
- Excessively deep hierarchies may be difficult to modify.

(Coad 91)

Heuristics for OO

- We need heuristics for the novel aspects of OO:
  - inventing classes;
  - inventing class hierarchies;
  - inventing class composition structures;
  - inventing message passing sequences.
Class Heuristics

A well-designed class:

- Provides a crisp abstraction of some thing drawn from the vocabulary of the problem domain or the solution domain.
- Embodies a small, well-defined set of responsibilities, and carries them all out very well.
- Provides a clear separation of the abstraction’s behaviour and its implementation.
- Is understandable and simple yet extensible and adaptable.

(Booch 96)

Inventing Classes (1)

- A class which has no operations should not be a class (it is probably an attribute). (Riel 1994)
- Avoid having too many services per class. Each class typically has no more than six or seven public services. (Coad 1991)
- If you can’t think of two good significantly different implementations, then a broader abstraction may be needed. (Coplien 1992)
Inventing Classes (2)

1. Each class should attempt to do its own validation. (Rising 1994)

2. A class should have easily identifiable and nameable behaviours. (Coplien 1992)

3. Beware classes which have only one instantiation in a system. (Riel 1994)

Inventing Class Hierarchies (1)

1. Abstract superclass rule (ASR) states: all superclasses must be abstract. (Hursch 1994)

2. Systems of around 100 classes should have a depth of inheritance hierarchy of approximately 7+2 levels. Excessively deep hierarchies may be difficult to modify. (Coad 91)

3. Inheritance should only be used for subtyping (as opposed to implementation inheritance) (Bar-David 1992)
Inventing Class Hierarchies (2)

1. Subclasses should not delete features of their superclasses. (Firesmith 1995)

2. Abstract classes should probably not have concrete ancestors. (Firesmith 1995)

3. If derived classes have no meaningful behaviour, attributes should have been used. (Rising 1994)

Inventing Composition Structures (1)

1. Containing classes should know what they contain, but contained classes should have no knowledge of who contains them. (Riel 1994)

2. Both classes in an association have equal status. If one class is subordinate to the other then aggregation or attribution should be used. (Rumbaugh 1996)

3. Designers should be clear about the precise relationship intended for each use of aggregation in a design. (Civello 1993)
Inventing Composition Structures (2)

1. When given the choice of using association or containment, chose containment. (Riel 1994)

2. Ensure that relationships among components of the system are as tenuous as possible. (Berard 1993)

3. The larger the number of other distinct system components with which an object must deal directly, usually the poorer is the design of the overall system. (Berard 1993)

Composition Versus Inheritance

- Both represent mechanisms for extending a class design
- Composition extends a class through encapsulation. A class may encapsulate one or more “child” objects
- Inheritance extends a class through a subtyping relationship. A new “subclass” inherits properties from an existing “superclass”
Composition

- The work of a parent object is delegated to its children
- It represents the major mechanism for extending the functionality of an existing class
- There are two types of composition relationship supported by the UML – aggregation and composition

Aggregation

```
Squadron
Aggregation

1

F16 Fighter
```

10..12

104
Aggregation

- Aggregation is where the lifetime of the “child” object(s) are not dependent on the parent
- In the previous example the fact that the squadron is being disbanded should not have a direct impact on the F16 Fighter objects!
- Aggregation should typically be used where composition implies “membership” or “seniority”
- e.g. Max is a citizen of the United Kingdom
- e.g. Joe works for Jess
- e.g. Michael J. Fox is in “Back to the Future”

Composition

- Diagram showing composition between Body and Lungs.
Composition

• Composition implies that the lifetimes of parent and child are tied together
• If the parent is deleted/destroyed/expunged with extreme prejudice so should the child objects
• Composition should be used where a “is a part of” relationship is required
• e.g. England is “a part of” the United Kingdom
• e.g. The brain is “a part of” a body
• e.g. An engine is “a part of” a car

Inheritance

• One of the defining relationships of the OO paradigm, and in fact is much overused
• It is a mechanism for extending an existing class (the superclass) by adding (or refining) attributes and methods and constructing a subclass.
• The subclass is considered to be a specialization of the existing superclass
• In other words a subclass adds or redefines new attributes and methods to those provided by an existing superclass
Inheritance (2)

- If a subclass inherits from a single superclass this is called (drum roll) … single inheritance!
- If it inherits from more than one superclass it is called … multiple inheritance
- IMHO: one of the best decisions taken by the developers of Java was to limit it to single inheritance ONLY
Inheritance – An Example

<table>
<thead>
<tr>
<th>Account</th>
<th>DepositAccount</th>
</tr>
</thead>
<tbody>
<tr>
<td>-balance: Money</td>
<td>-interestRate: Percent</td>
</tr>
<tr>
<td>-accountHolder: String</td>
<td>creditInterest ()</td>
</tr>
<tr>
<td>getBalance ()</td>
<td>debitInterest ()</td>
</tr>
<tr>
<td>setBalance ()</td>
<td></td>
</tr>
<tr>
<td>getAccountHolder()</td>
<td></td>
</tr>
<tr>
<td>setAccountHolder()</td>
<td></td>
</tr>
</tbody>
</table>

Benefits of Inheritance

- When everything works out it can be very cool
- It strongly promotes reusability by building upon existing (and one hopes reliable) superclass designs
- It promotes substitutability, subclass objects can be used in the same places as superclass objects (as long as the superclass interface only is invoked)
- This is immensely powerful if the subclass redefines existing superclass methods with new/improved versions of the code
**Why Inheritance can be Bad!**

- Inheritance promotes *weak* encapsulation between a subclass and superclass. This can lead to unpleasant side effects if the subclass uses superclass methods and operations in an inappropriate way!
- It is doubly unfortunate as strong encapsulation really is the fundamental design tenet of the whole OO philosophy!

---

**When to Use Inheritance**

- When the subclass is “a special kind of” not when it “is a role played by”
- The subclass never needs to transmute its class
- A subclass extends the superclass rather than overrides or nullifies it
- Does not simply subclass a nifty utility class
- Expresses special types of roles, devices or transactions
Otherwise…

- Use composition!
- Inheritance can add an enormous degree of semantic complexity to a design
- In recent years it has been recognized that composition is far more appropriate for extending a class design than inheritance.

An Illustrative Example

```
Person
  └── Travel Agent
  └── Customer
```
Analysis (1)

• At first glance it looks fine – a Person, subclassed into two roles – Travel Agent and Customer
• However … we introduce additional complexity if our travel agent (quite reasonably) decides to buy a vacation…

An Illustrative Example (2)
Analysis (2)

• Now we have a nasty situation where we are forced to use multiple inheritance to combine travel agent and customer into a single abstraction
• The only other possibility is to copy the values in a travel agent object into a customer object
• This is a potentially “lossy” copy as the travel agent specific information will be lost (or will need to be maintained separately)
• Either way you have an unfortunate mess

The Inheritance Checklist

• The subclass(es) are “a kind of” not “a role played by” – fail. These are roles.
• Never needs to transmute its class – fail. When our travel agent needs to go on vacation his class needs to change!
• Extends rather than overrides or nullifies – pass. Ok good to go here.
• Does not subclass utility classes – ok pass.
• Expresses special kinds of roles, transactions or devices. Deceptive fail. Is Person a role in the first place?
A Composition-Based Solution

Composition : Your Friend 😊

- This is an example of a role-player pattern
- A person can encapsulate a number of role objects – in a given context it can be one role or another.
- New roles can be added or removed at will
- The Person object itself is never forced to change class
- All interaction between the roles is encapsulated inside the player. The context-switching between roles is managed by the player object and isolated from external clients.
Message Passing Sequences (1)

Avoid having a large, central object which controls most of the functionality. (Riel 1994)

Message passing may be used to:
  • request a corresponding service
  • provide notification of an event
  • provide data (actual parameters). (Firesmith 1993)

Minimise object collaboration. In most cases objects should only collaborate with three to five other objects. (Coad 1991)

Message Passing Sequences (2)

1. Variables should only be accessed or modified through the sending of messages to, or the invoking of, the accessing methods. (Wirfs-Brock and Wilkerson 1989)

2. Services should do some work, not just pass the buck to another service. (Love 1991)

3. Keep protocols as simple as possible, have no more than 6 data parameters per method. (Rising 1994)
Miscellaneous Heuristics (1)

Classes should be grouped into defined collections of up to twenty classes that have restricted visibility from other classes in the system. (Love 1991)

Do not use protected data, it weakens data hiding. Use protected access functions instead. All data in the base class should be private. (Riel 1994)

An object removed from the context of the immediate application, should, in isolation, still represent a coherent and complete object-oriented concept. (Berard 1993)

Miscellaneous Heuristics (2)

Use polymorphism rather than switch/case statements for decisions based on type. (Firesmith 1995)

Model the real world whenever convenient since this will provide a more natural platform for maintenance personnel. (Riel 1994)

Great classes are essentially interface specifications embodied as abstract classes. We don’t create the abstract class because it is evident in the domain - we create it to control the impact of future changes in the domain. (Haythorn 1994)
Evaluation in an Iterative Process

Lifecycle
Various heuristics are applicable at analysis time or design time and some at both.

Subjectivity
Many heuristics are reliant upon interpretation

Specificity
Certain heuristics are language dependent such as those that concern the use of C++ ‘friend’ classes.

Issues in the Use of Heuristics

Heuristics

- The role of heuristics in product reviews
- Heuristic ownership
- Resolving the trade-offs
- Providing training in the heuristics
- What makes any heuristic convincing?
References

Henderson-Sellers, B. *Object-Oriented Metrics: measures of complexity* Prentice Hall 1996

Lorenz, M and Kidd, J. *Object-Oriented Software Metrics* Prentice Hall 1994


More References


The Law of Demeter

Do not refer to a class $C$ in a method $m$ unless $C$ is the type of

- an instance variable in $m$’s class definition,
- an argument of $m$,
- an object created in $m$,
- a global variable.


The Law of Demeter Example

// violates the law
aBoard[i][j].getPiece.setColour(BLACK)

// complies with the law
aBoard.setPieceColour(i, j, BLACK);
Metric Based Design Assessment

OO metrics have looked at a range of issues.

- Reuse
- Size
- Product
- Process

We concentrate on product metrics.

Measurement Theory

**Nominal Scale** - give a name or attach a label
   - Bus numbers, Football squad numbers

**Ordinal Scale** - permit ordering
   - Agree / Neutral / Disagree
   - Cannot have a mean

**Interval Scale** - the ‘distance’ between items matters
   - The Celsius temperature scale. Can have a mean

**Ratio Scale** - like an interval scale but zero counts
   - Mass, length and time

**Absolute Scale** - are absolute counts
   - Lines of code
   - Allow a full range of stats
Assessing Structure

**Coupling**
- countable
- ranges from data coupling to code coupling

**Cohesion**
- not countable
- ranges from functional to coincidental

**Complexity**
- measured using Halstead and McCabe

---

How Big is a Class?

We start by asking for the value of NOM, the number of methods in a class.

Do we consider private methods which are only used in the class or do we only count the ones in the public interface?

Do we count instance methods or do we count both instance and class (static) methods?

What does this question look like in the presence of inheritance and polymorphism?
**So How Big is a System?**

System size =
- NOM Parent +
- NOM Child +
- NOM GrandChild +
- NOM Relative

**MOOD - the Project**

Metrics for Object Oriented Design (MOOD) Brito e Abreu

MOODKIT is a tool to automatically extract the metrics from C++ and Eiffel (and Java)

The metrics are all presented as quotients (percentages)

The metrics are presented in a broad framework which supports both product metrics and process metrics.
MOOD - the Metrics

Method Hiding Factor (MHF)
The sum of the invisibilities/the total number of methods

Attribute Hiding Factor (AHF)
The sum of the invisibilities/the total number of methods

Method Inheritance Factor (MIF)
The sum of inherited methods/the total number of methods

Attribute Inheritance Factor (AIF)
The sum of inherited attributes/the total number of attributes

Polymorphism Factor (PF)
The actual number of polymorphic situations/the maximum number

Coupling Factor (CF)
The actual(non-inheritance) couplings/the maximum possible

MOOD - Some Results

<table>
<thead>
<tr>
<th>Microsoft Foundation Classes</th>
<th>MFC</th>
<th>GNU</th>
<th>ET++</th>
<th>Motif</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>128</td>
<td>84</td>
<td>262</td>
<td>35</td>
<td>595</td>
</tr>
<tr>
<td>Methods</td>
<td>3080</td>
<td>1478</td>
<td>4812</td>
<td>199</td>
<td>10417</td>
</tr>
<tr>
<td>Attrib.</td>
<td>608</td>
<td>151</td>
<td>980</td>
<td>76</td>
<td>1940</td>
</tr>
<tr>
<td>LOC</td>
<td>74895</td>
<td>15960</td>
<td>55022</td>
<td>4884</td>
<td>163556</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>MFC</th>
<th>GNU</th>
<th>ET++</th>
<th>Motif</th>
</tr>
</thead>
<tbody>
<tr>
<td>MHF</td>
<td>24.6%</td>
<td>13.3%</td>
<td>9.6%</td>
<td>39.2%</td>
</tr>
<tr>
<td>AHF</td>
<td>68.4%</td>
<td>84.1%</td>
<td>69.4%</td>
<td>100.0%</td>
</tr>
<tr>
<td>MIF</td>
<td>83.2%</td>
<td>63.1%</td>
<td>83.9%</td>
<td>64.3%</td>
</tr>
<tr>
<td>AIF</td>
<td>59.6%</td>
<td>62.6%</td>
<td>51.8%</td>
<td>50.3%</td>
</tr>
<tr>
<td>CF</td>
<td>9.0%</td>
<td>2.8%</td>
<td>7.7%</td>
<td>7.6%</td>
</tr>
<tr>
<td>PF</td>
<td>2.7%</td>
<td>3.5%</td>
<td>4.5%</td>
<td>9.8%</td>
</tr>
</tbody>
</table>
Chidamber & Kemerer


- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number Of Children (NOC)
- Coupling Between Object classes (CBO)
- Response For a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

Weighted Methods per Class

The number of methods and the complexity of methods involved is an indicator of how much time and effort is required to develop and maintain a class.

The larger the number of methods in a class the greater the potential impact on children, since children will inherit all the methods defined in the class.

Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.
Depth of Inheritance Tree

The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior.

Deeper trees constitute greater design complexity, since more methods and classes are involved.

The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

Number of Children

Greater the number of children, greater the reuse, since inheritance promotes reuse.

Greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing.

The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.
Coupling Between Object classes

Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent an object is, the easier it is to reuse it in another application.

In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.

A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object coupling, the more rigorous the testing needs to be.

Response For a Class

If a large number of methods can be invoked in response to a message, the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester.

The larger the number of methods that can be invoked from a class, the greater the complexity of the class.

A worst case value for possible responses will assist in appropriate allocation of testing time.
Lack of Cohesion in Methods

Cohesiveness of methods within a class is desirable, since it promotes encapsulation.

Lack of cohesion implies classes should probably be split into two or more sub/classes.

Any measure of disparateness of methods helps identify flaws in the design of classes.

Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.