Software Architecture

Summary of the course in spring 2010 by Bertrand Meyer

Stefan Heule

2010-06-02

Table of Contents

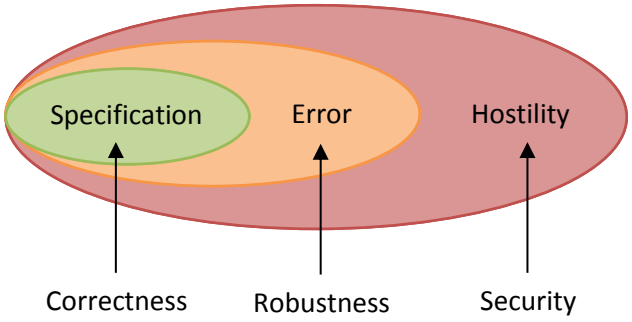
1 Introduction

1.1 Definitions

- **Software architecture**: The decomposition of software systems into modules
  - Primary criteria: extendibility and reusability
- **Software engineering**: The application of engineering principles and techniques, based on mathematics, to the development and operation of possibly large software systems satisfying defined standards of quality.

1.1.1 Software quality factors

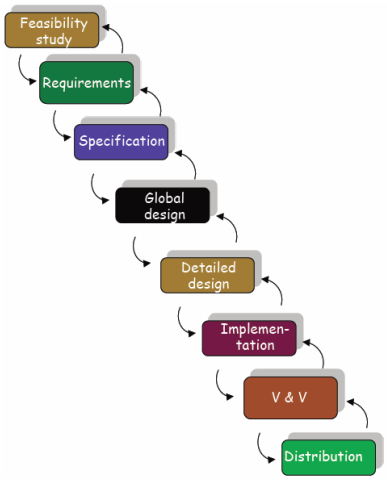
- Product
  - Immediate
    - Correctness
    - Robustness
    - Security
    - Ease of use
    - Ease of learning
    - Efficiency
  - Long-term
    - Extendibility
    - Reusability
    - Portability
- Process
  - Timeliness
  - Cost-effectiveness



1.1.2 Task groups of software engineering

- Describe
  - Requirements, design specification, documentation
- Implement
  - Design, programming
- Assess
  - Verification and validation, especially testing
    - Validation: Ensure that the functionality built works properly (often internal process)
    - Verification: Ensure that the right functionality is being build (often done with the customer)
- Manage
  - Plans, schedules, communication, reviews
- Operate
  - Deployment, installation

2 The software lifecycle

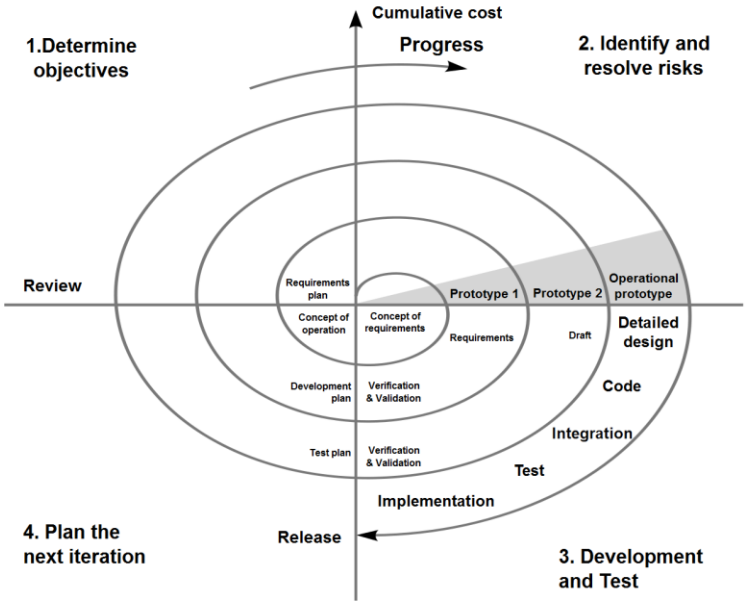


2.1 The waterfall model

- Arguments for the waterfall
  - The activities are necessary
  - The order is the right one
- Problems
  - Late appearance of actual code
  - Lack of support for requirements change
  - Lack of support for maintenance activities
  - Highly synchronous model

2.2 The spiral model

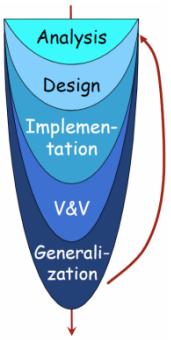
- Apply a waterfall-like approach to successive prototypes.



- Problems include “plan to throw one away”, which is bad advice. There is also the risk of shipping the prototype.

2.3 Seamless, incremental development

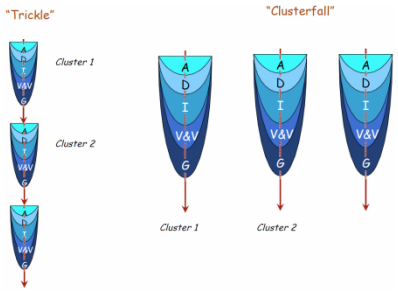
- Seamless development
  - Single set of notation, tools, concepts and principles
  - Continuous, incremental development
  - Keep model, documentation and implementation consistent
- Reversibility: It is possible to go back and forth.
- Generalization: prepare for reuse
  - Remove built-in limits
  - Remove dependencies
  - Improve documentation, contracts, ..



- Few companies have the guts to provide budget for this

2.3.1 The cluster model

- The seamless development can be applied to multiple clusters. Two extremes variants of this are shown in the image.



3 Requirements

3.1 Definitions

- **Requirements engineering** is the process of defining the service that a customer requires from a system and the constraints under which it operates.
- **A requirement** is a statement of desired behavior for a system or a constraint on a system.
- **The requirements** for a system are the collection of all such individual requirements.

3.2 Overview

- The costs to correct a defect grow exponentially with the time spent on the project. Correcting a bug in the requirements phase is relatively cheap; during the development testing it is still acceptable while defects found during operation are extremely expensive.

3.2.1 Goals of performing requirements

- Understand the problem
- Decide what the system should do and what it should not do
- Provide a basis for development and validation/verification, especially testing
- Identify stakeholders
- **Requirements determine test plan**
- Quality goals for requirements
    - Justified
    - Correct
    - Complete
    - Consistent
    - Unambiguous
    - Feasible
    - Abstract
    - Traceable
    - Delimited
    - Interfaced
    - Readable
    - Modifiable
    - Verifiable
    - Prioritized
    - Endorsed

3.2.2 Difficulties of requirements

- Natural language and its imprecision
- Formal techniques and their abstraction
- Users and their vagueness
- Customers and their demands
- Committing too early to an implementation: **over-specification**
- Missing parts of the problem: **under-specification**

3.2.3 Two parts of requirements

- Machine specification and domain properties => requirements
    - Machine specification: Desired properties of the machine
    - Domain properties: Outside constraints
    - Requirements: Desired system behavior

3.2.4 Standards and methods

- IEEE Recommended Practice for Software Requirements Specification, IEEE 830-1998

3.2.5 Requirements under agile methods

- Requirements are taken into account as defined at the particular time considered.
- Requirements are largely embedded in test cases
- Benefits
    - Test plan will be directly available
    - Customer involvement
- Risks
    - Change may be difficult
    - Structure may not be right
    - Test only cover the foreseen cases

3.3 Requirements elicitation

- Before elicitation: at a minimum
    - Overall project description
    - Draft glossary
- Overall scheme
    - Identify stakeholders
    - Gather wish list of each category
    - Document and refine wish list
    - Integrate, reconcile and verify list
    - Define priorities
    - Add any missing elements and non-functional requirements
- After elicitation
    - Examine requirements from the viewpoint of requirements quality factors, especially consistency and completeness
    - Finalize scope of project

3.4 Use cases and scenarios

- A **use case** describes how to achieve a single business goal or task through the interactions between external actors and the system
    - **Actors** are the interacting parties outside of the system, e.g. user classes, other systems
    - **Scenario** is an instance of a use case representing a single path through the use case
- A good use case must
    - Single business task goal
    - Describe alternatives, failures and exceptional behavior
    - Treat system as a black box
    - Not implementation specific
    - Captures **who** (actor) does **what** (interaction) **why** (goal)
- Discussion

- Use cases are a tool for requirement elicitation but insufficient to define requirements
  - Not abstract enough
  - Too specific
  - Do not support evolution

4 Modularity and abstract data types

4.1 Modularity

- General goal: Ensure that software systems are structured into units (modules) chosen to favor:
  - Extendibility
  - Reusability
  - Maintainability
- Principles
  - **Decomposability**
    - The method helps decompose complex problems into subproblems
  - **Composabilty**
    - The method favors the production of software elements that may be freely combined with each other to produce new software
    - **Few interfaces principle**: Every module communicates with as few others as possible
    - **Small interface principle**: If two modules communicate, they exchange as little information as possible
    - **Explicit interface principle**: Whenever two modules communicate, this is clear from the text of one or both of them
  - **Continuity**
    - The method ensures that small changes in the specification yield small changes in the architecture.
  - **Information hiding**
    - The designer of every module must select a subset of the module's properties as official information about the module, to be made available of client modules
    - This is justified due to continuity and decomposability
  - **Uniform access principle**
    - It does not matter to the client whether you look up or compute a value
  - **The open-closed principle**
    - Modules should be both open and closed.
    - Open module: may be extended
    - Closed module: usable by clients. May be approved, baselined and compiled
  - **The single choice principle**
    - Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list

4.2 Abstract data types

- A formal way of describing data structures with the following benefits
  - Modular, precise description of a wide range of problems
  - Enables proofs
  - Basis for object technology
  - Basis for object-oriented requirements

4.2.1 Overview

An abstract data type specification consists of four parts

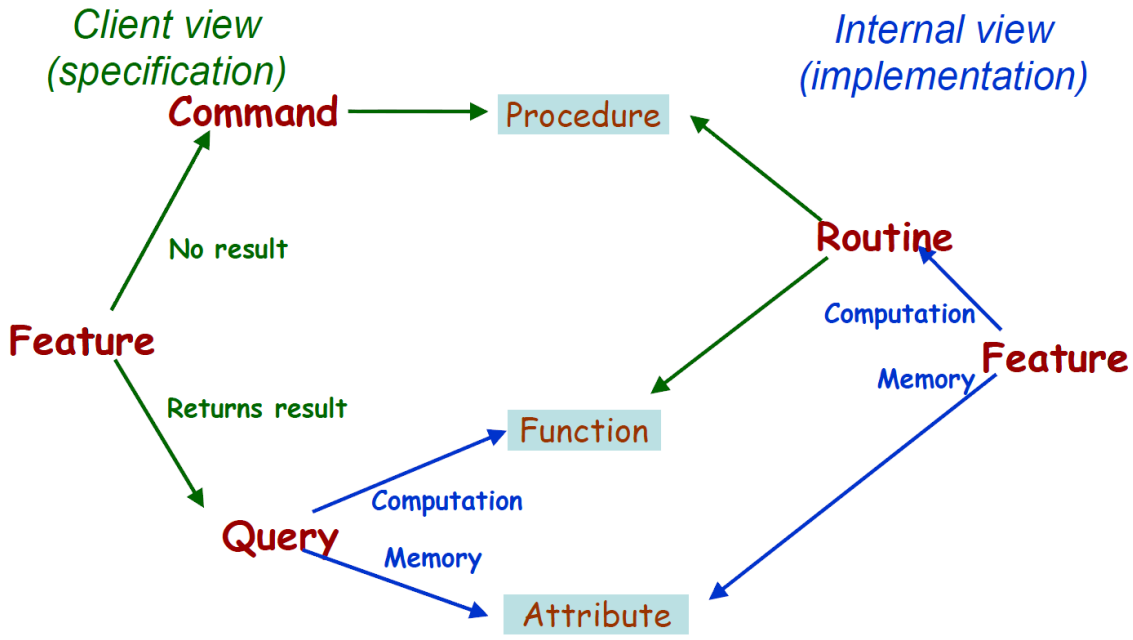
- Types
  - An abstract data type is a set of objects, e.g. $STACK[G]$, possible using genericity.
- Functions
  - List of functions as signatures that are available for the types, e.g.
    - $put: STACK[G] \times G \rightarrow STACK[G]$
    - new: STACK[G]
  - Function in the mathematical sense, absolutely no side-effects
  - Function categories of an ADT specification for type T
    - **Creator function**: functions where T only appear on the right-hand side of the (possibly omitted) arrow, e.g. new
    - **Query function**: functions where T only appear on the left-hand side of the arrow, e.g. $empty$
    - **Command function**: functions where T appears on both sides of the arrow, e.g. put
- Axioms
  - ADTs are free of implementation details, therefore implicit definitions are used to specify functional behavior, e.g.
    - $item\big(put(s,x)\big) = x$
- Preconditions
  - Some functions are not always applicable, i.e. they are partial functions. This is expressed using preconditions, e.g.
    - $item(s: STACK[G])$ requires not $empty(s)$
  - Such partial functions are defined (see "Functions" above) using a special arrow: $\nrightarrow$

4.2.2 Sufficient completeness

- **Correct ADT expression**
  - Let $f(x_1, x_2, \dots, x_n)$ be a well-formed expression involving one or more function on a certain ADT. This expression is correct if and only if all the x_i are (recursively) correct, and their values satisfy the precondition of f.
- **Sufficient completeness**
  - An ADT specification for a type T is sufficiently complete if and only if the axioms of the theory make it possible to solve the following two problems for any well-formed expression e:
    - Determine whether e is correct
    - If e is a query expression and has been shown to be correct, express the value of e under a form not involving any value of type T.
- **ADT consistency**
  - An ADT specification is consistent if and only if, for any well-formed query expression e, the axioms make it possible to infer at most one value for e.

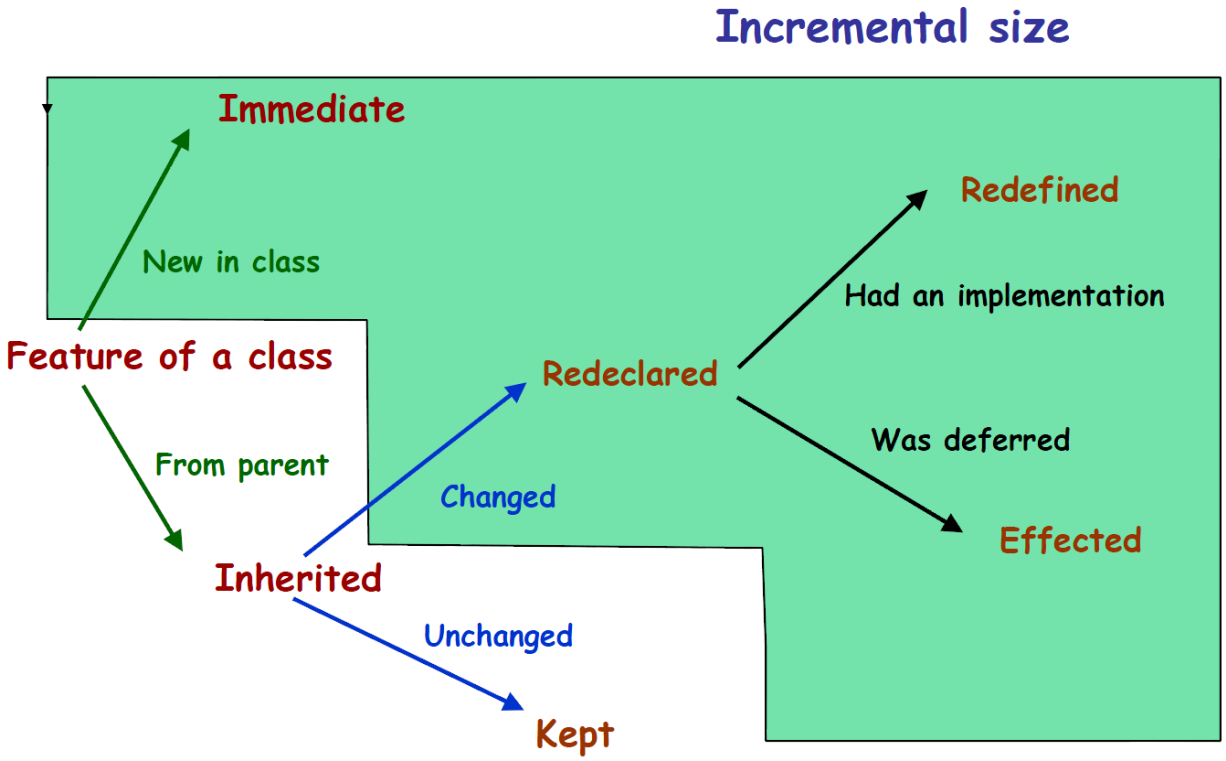
5 Designing for reuse (a.k.a. random things)

- A **component** is a program element such that
  - It may be used by other program elements
  - Its authors need not know about the clients
  - Clients' authors need only know what the component's author tells them
- Why reuse?
  - Consumer view
    - Faster time to market
    - Guaranteed quality
    - Ease of maintenance
  - Producer view
    - Standardization of software practices
    - Preservation of know-how
- Levels of reusability
  - 0 - Usable in some program
  - 1 - Usable by program written by the same author
  - 2 - Usable within a group/company
  - 3 - Usable within a community
  - 4 - Usable by anyone
- Remainder of feature classifications



- **Referential transparency**
  - If two expression have equal value, one may be substituted for the other in any context where that other is valid.
    - If $a = b$, then $f(a) = f(b)$, i.e. this prohibits functions with side effects.
- How big should a class be?

- The first question is how to measure class size. Candidates are: source lines, number of features. For features we have:
  - With respect to information hiding
    - Internal size: includes non-exported features
    - External size: only exported features
  - With respect to inheritance
    - Immediate size: includes new (immediate) features only
    - Flat size: includes immediate and inherited features
    - Incremental size: includes immediate and redeclared features



  - Shopping list approach: if a feature may be useful, it probably is. No need to limit classes to "atomic" features.

- Language and library
  - The language should be small
  - The library should provide as many useful facilities as possible
- Arguments are of one of the following two types:
  - **Operands**: values on which the feature will operate
  - **Options**: modes that govern how the feature will operate
    - Criteria to recognize options:
      - There is a reasonable default value
      - During the evolution of a class, operands will normally remain the same, but options may be added
  - **Option-operand principle**
    - Only operands should appear as arguments of a feature

        - Option values have defaults and can be set to specific values using setter procedures.
- Naming
    - Use meaningful variable names
    - Achieve consistency by systematically using a set of standardized names
    - Emphasis commonality over differences. Differences will be captures by
        - Signatures (number and types of arguments)
        - Assertions
        - Comments
    - Grammatical rules
        - Procedures (commands): verbs in infinite form, e.g. *make*, *put*, *display*
        - Boolean queries: adjectives, e.g. *is_full, is_first*
        - Other queries: nouns and adjectives, e.g. *count*, *error_window*

6 Design patterns

6.1 Introduction

- A pattern is a three-part rule, which expresses a *relation* between a certain *context*, a *problem* and a *solution*.
- Benefits of design patterns
  - Capture the knowledge of experienced developers
  - Newcomers can learn and apply patterns
  - Yield better software structure
- A **design pattern** is an architectural scheme – a certain organization of classes and features – that provides applications with a standardized solution to a common problem.
  - Design pattern are not reusable in general: Each pattern describes a problem that occurs over and over again, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice.

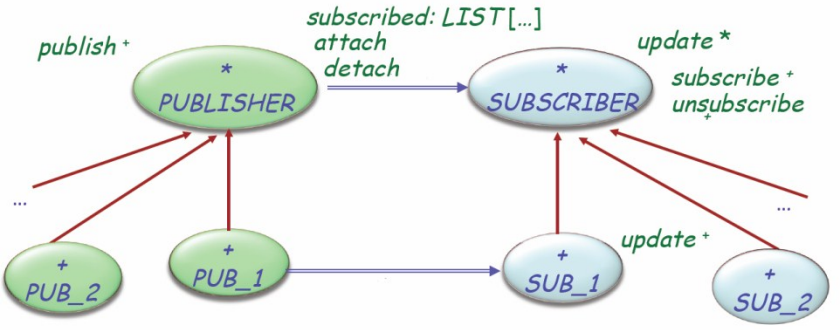
6.2 Pattern categorization

- Creational patterns
  - Hide the creation process of objects
  - Hide the concrete type of these objects
  - Allow dynamic and static configuration of the system
- Structural patterns
- Behavioral patterns

6.3 Patterns

6.3.1 Observer pattern

- Intent:
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



- Publisher keeps a secret list of observers
- Subscribers implement a routine update that will be called on all subscribers if an event is published.
- Participants

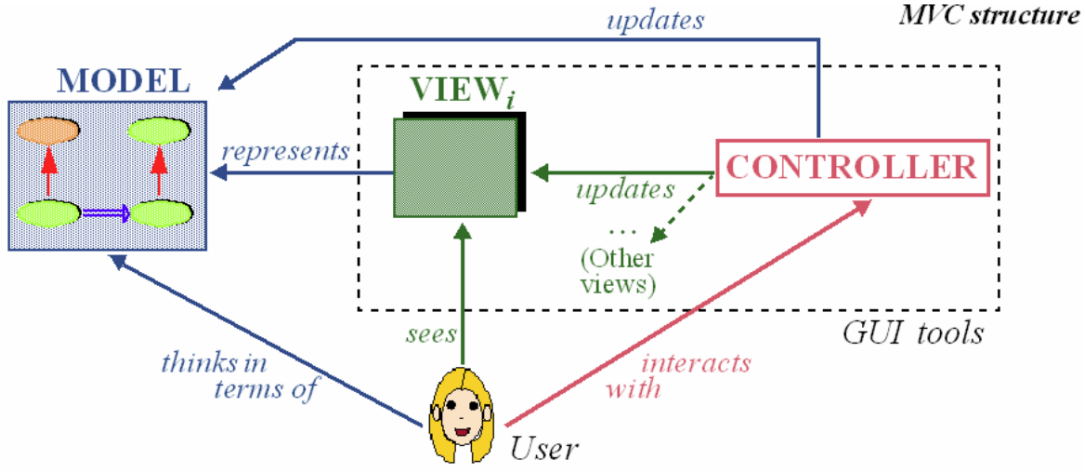
- Publisher
  - Knows its subscribers
  - Provides an interface for attaching and detaching subscribers
- Subscriber
  - Defines an updating interface for objects
- Concrete publisher
  - Stores state of interest and sends notifications to all subscribers when its state changes
- Concrete subscriber
  - Maintains a reference to a publisher object
  - Stores state that should stay consistent with the publisher's

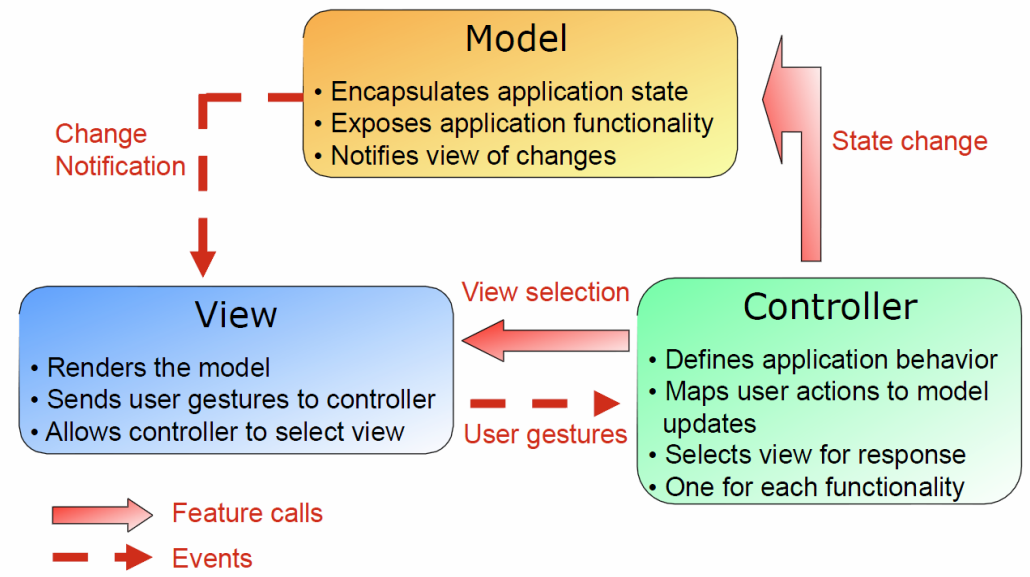
- Problems with this (simple approach)
  - Subscribers may only subscribe one operation to at most one publisher (!)
  - Event arguments are tricky to handle
  - Subscriber knows publisher
  - Not reusable

6.3.2 Event library

- Intent
  - Improved version of the observer pattern
- Each event type will be an object (e.g. left click)
- Context is an object, usually representing a user interface element (e.g. button)
- Action is an agent representing a routine
- One basic class EVENT_TYPE
  - *publish* to publish events
    - *click.publish ([x,y])*
  - *subscribe* to subscribe to events
    - *click.subscribe (agent find_station)*

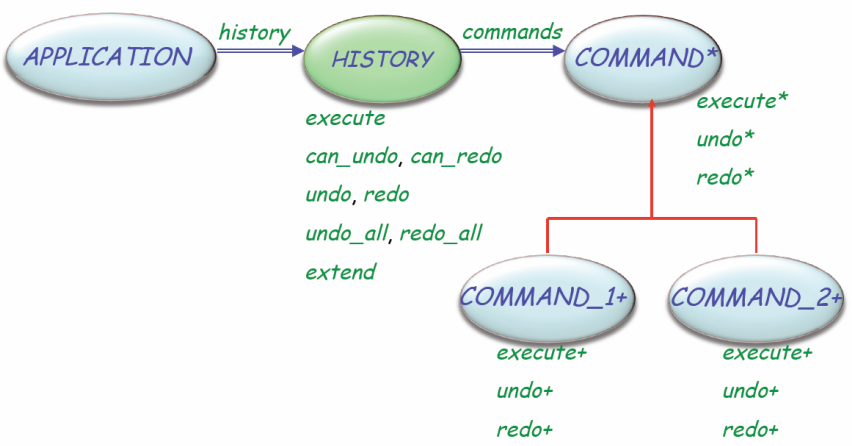
6.3.3 Model-view-controller





6.3.4 Command pattern (undo/redo)

- Intent
  - Way to implement an undo-redo mechanism, e.g. in text editors
- A history is kept as a list of COMMANDs
  - A command object includes information about one execution of a command by the user, sufficient to execute the command, and cancel it later

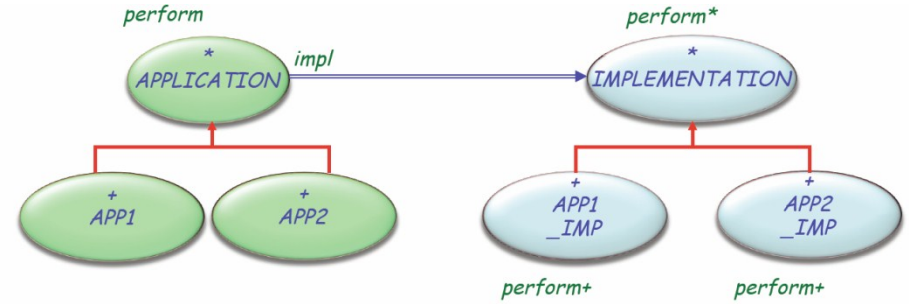


- It is also possible to use pairs of PROCEDUREs to implement the history. This reduces the amount of (possibly small) classes
- Consequences
  - Decouple the object that invokes the operation from the one that knows how to perform it
  - Commands are first-class objects that can be manipulated and extended like any other object
  - Possible to assemble commands into composite commands
  - Easy to add new commands, no existing classes have to be changed

- Participants
  - Command
    - Declare interface for executing operation
  - Concrete command
    - Define a binding between a receiver object and an action
    - Implements *execute* by invoking the corresponding operations on receiver
  - Client
    - Creates a concrete command object and sets its receiver
  - Invoker
    - Asks the command to carry out the request
  - Receiver
    - Knows how to perform the operations associated with carrying out a request. Any class may serve as a receiver

6.3.5 Bridge

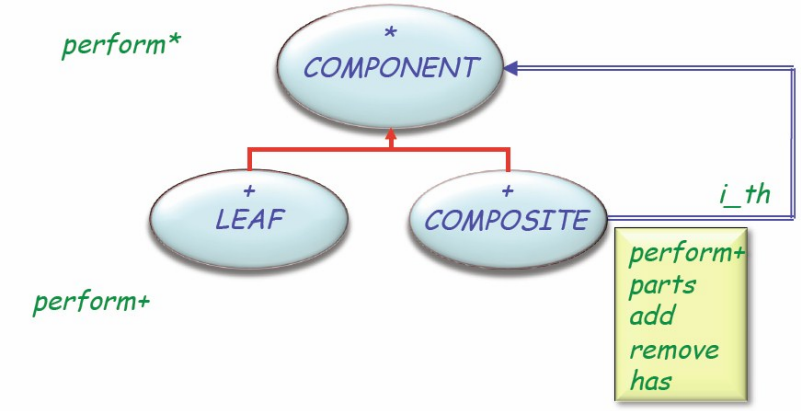
- Intent
  - Decouple an abstraction from its implementation so that the two can vary. It separates the class interface (visible to the clients) from the implementation (that may change later).



- Applications are created by providing an implementation. It is then possible to call *perform* on these objects.
- Advantages
  - No permanent binding between abstraction and implementation
  - Abstraction and implementation extendible by subclassing
  - Implementation changes have no impact on clients
  - Implementation of an abstraction completely hidden from clients

6.3.6 Composite pattern

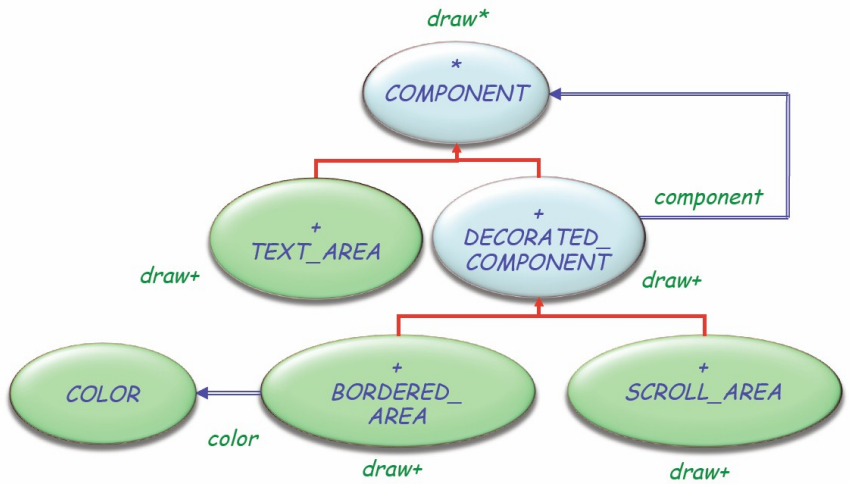
- Intent
  - Way to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



- Advantages
  - Represent part-whole hierarchies
  - Treat compositions and individual objects uniformly

6.3.7 Decorator pattern

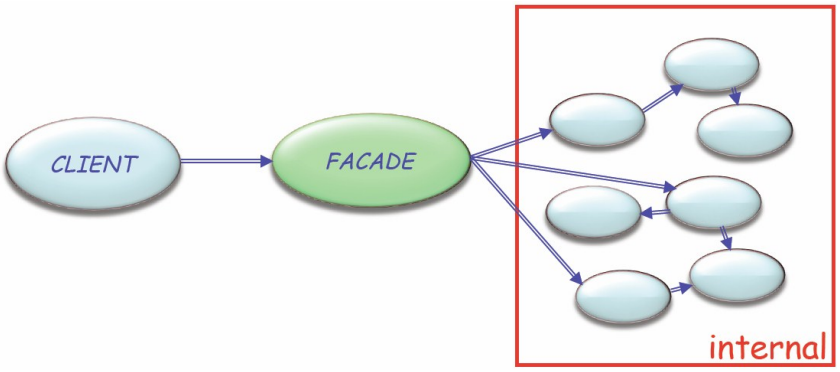
- Intent
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Example



- Advantages
  - Add responsibilities dynamically
  - Responsibilities can be withdrawn
  - Omit explosion of subclasses to support combinations of responsibilities

6.3.8 Façade

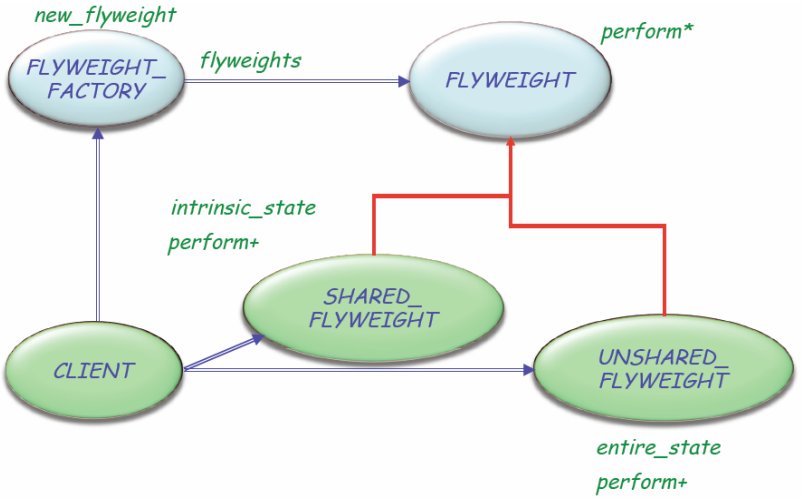
- Intent
  - Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.



- Advantages
  - Simple interface to complex subsystem
  - Decouples clients from the subsystem and fosters portability
  - Can be used to layer subsystems by using façades to define entry points for each subsystem level

6.3.9 Flyweight pattern

- Intent
  - Use sharing to support large numbers of fine-grained objects efficiently
- Examples
  - Instead of having a line object for every line, there is only one line per color
  - One flyweight per character code in a text processing application
- Basic distinction
  - Intrinsic properties of the state are stored in the flyweight (e.g. color)
  - Extrinsic properties are stored in the "context" for each use (e.g. coordinates of the line)



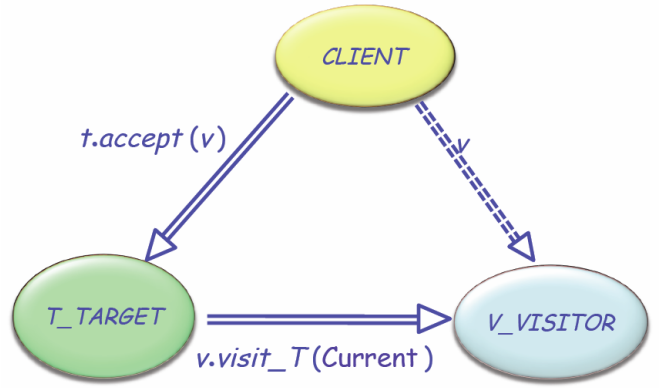
- Participants
  - FLYWEIGTH: Offers a service *perform* to which the extrinsic characteristic will be passed.
  - SHARED_FLYWEIGHT: Adds storage for intrinsic characteristics
  - UNSHARED_FLYWEIGHT: Not all flyweight need to be shared
  - FACTORY: Creates and manages flyweight objects

- CLIENT: Maintains a reference to flyweight, and computes or stores the extrinsic characteristics of flyweight

- Advantages
  - Reduce storage for large number of objects
    - By reducing the number of objects using shared objects
    - By reducing the replication of intrinsic state
    - By computing (rather than storing) extrinsic state

6.3.10 Visitor pattern

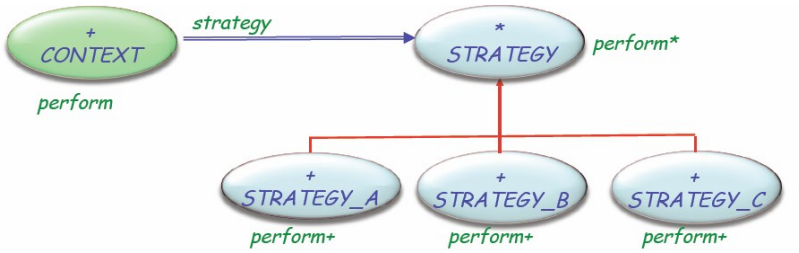
- Intent
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Often it is wishful to have multiple operations that can be performed on a set of classes (class hierarchies). Adding these operation directly to the class hierarchy can clutter the code, and makes it very inconvenient to add new operations. Visitor tries to solve this.



- Participants
  - Visitor: General notion of a visitor
  - Concrete visitor: Specific visit operation, applicable to all target elements.
  - Target: General notation of a visitable element.
  - Concrete target: Specific visitable element
- Consequences
  - Adding a new operation is easy by gathering related operations and separation unrelated ones
  - No object tests, which leads to better type checking
  - Adding a new concrete element is hard

6.3.11 Strategy

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Example: selecting a sorting algorithm on-the-fly



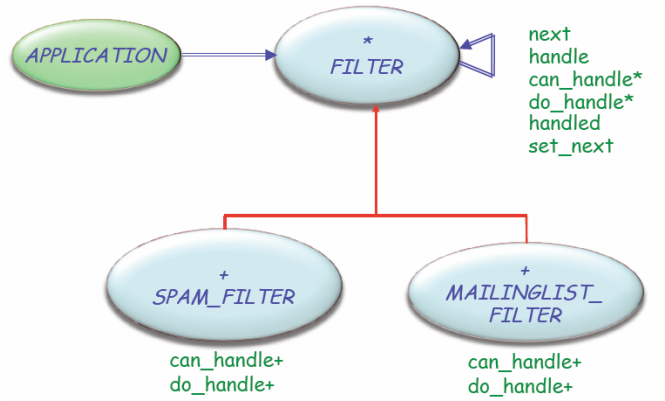
- Consequences
  - Provides alternative implementation without conditional instruction (e.g. using a parameter to specify the sorting mechanism to be used)
  - Clients must be aware of different strategies
  - Communication overhead between strategy and context
  - Increased number of objects
- Participants
  - Strategy declares an interface common to all supported algorithms
  - Concrete strategy implements an algorithm
  - Context is configured with a concrete strategy and keeps a reference to a strategy object.

6.3.11.1 Difference to the bridge pattern

- The bridge pattern and the strategy pattern are very similar. However, their intent is different:
  - Strategy: you have more ways for doing an operation; with strategy you can choice the algorithm at run-time and you can modify a single Strategy without a lot of side-effects at compile-time;
  - Bridge: you can split the hierarchy of interface and class join him with an abstract reference

6.3.12 Chain of responsibility

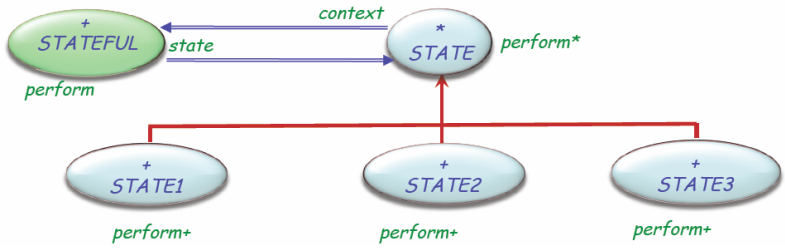
- Intent
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- If a filter can handle the request, it will. Otherwise it will pass it on to the next filter, until it drops of the chain.



- Consequences
  - Reduced coupling
    - An object only has to know that a request will be handled "appropriately". Both the receiver and sender have no explicit knowledge of each other.
  - Added flexibility in assigning responsibilities to objects
    - Ability to add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time
  - Receipt is not guaranteed
    - The request can fall off the end of the chain without being handled

6.3.13 State pattern

- Intent
  - Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Example: Mouse actions in a painting application have different behavior depending on the current tool (e.g. pen tool, selection tool,..)



- Consequences
  - The pattern localizes state-specific behavior and partitions behavior for different states
  - Make state transitions explicit
  - State objects can be shared
- Participants
  - Context
    - Defines the interface of interest to clients
    - Maintain an instance of a concrete state subclass that defines the current state
  - State

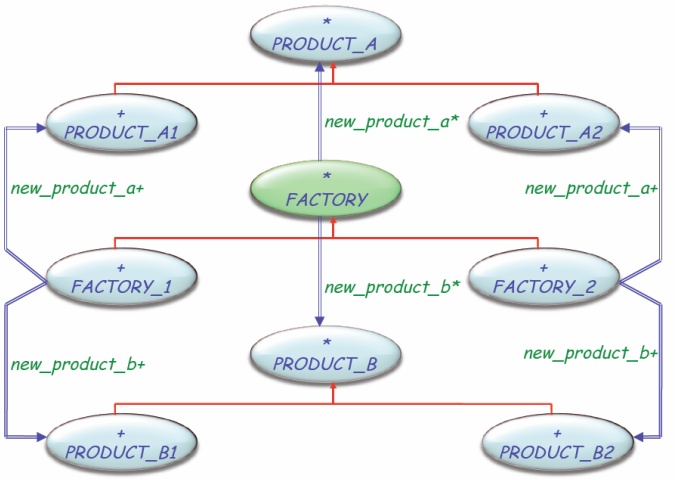
        - Defines an interface for encapsulating the behavior associated with a particular state of the context
    - Concrete state
        - Each subclass implements a behavior associated with a state of the context

6.3.14 Factory method pattern

- Intent
    - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses.
- Instead of *create {T} x.make* the user now writes *x := factory.new_t*
    - However, this is not just a syntactic replacement, as T could be a deferred class. In this case, the first statement would not be possible.

6.3.15 Abstract factory pattern

- Intent
    - Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- Example: widget toolkit in EiffelVision or Java Swing to provide different look and feels. Family of widgets (e.g. scroll bars, buttons, ..); want to allow changing look and feel.
    - Most parts of the system do not need to know about what look and feel is used
    - Creation of widget objects should not be distributed



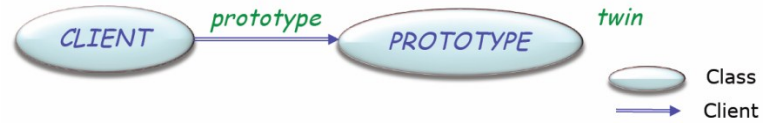
- Reasons to use abstract factory
    - Most parts of the system should be independent of how its objects are created, represented and collaborate
    - The system needs to be configured with one of multiple families
    - A family of objects is to be designed and only used together
    - You want to support a whole palette of products, but only show the public interface
- Properties
    - Isolate concrete classes
    - Make exchanging product families easy

- Promote consistency among products
- Support new kinds of products is difficult

- Factory method vs. abstract factory
  - Factory method
    - Create one object
    - Works at the routine level
    - Helps a class perform an operation, which requires creating an object
  - Abstract factory
    - Creates families of objects
    - Works at the class level
    - Uses factory methods (e.g. new_button is a factory method)

6.3.16 Prototype pattern

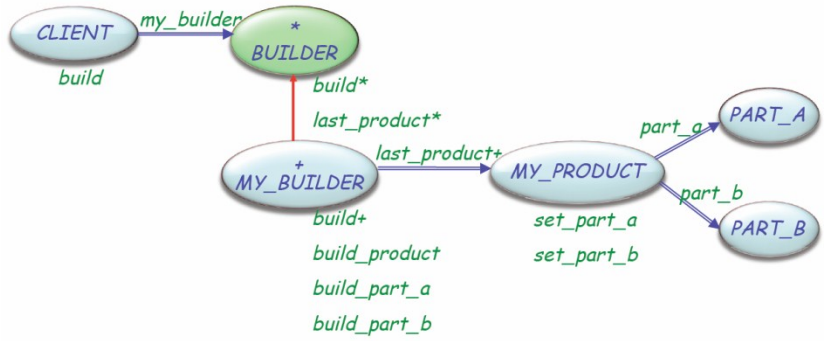
- Intent
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype



- Cloning in different languages
  - Java: classes must implement the interface *Clonable*.
  - C#: classes must implement the interface *IClonable.*

6.3.17 Builder pattern

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



6.3.17.1 *Difference to the factory pattern*

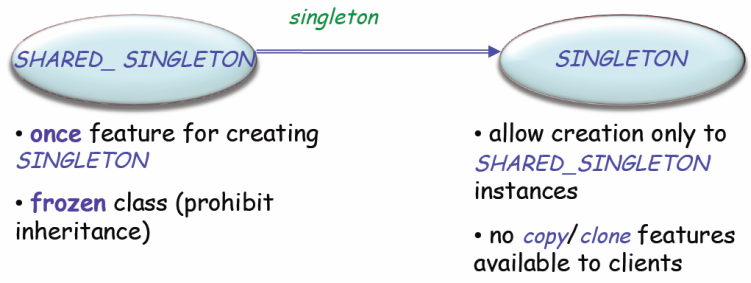
- The builder pattern is only in a limited way similar to factory pattern.
  - The factor pattern defers the choice of what concrete type of object to make until run time.

    - The builder pattern encapsulates the logic of how to put together a complex object so that the client just requests a configuration and the builder directs the logic of building it.
- The factory is concerned with **what** is made, the builder with **how** it is made.

6.3.18 Singleton

- Intent
    - Way to ensure a class only has one instance, and to provide a global point of access to it.

- Eiffel
    - Once routines ensure that the instructions are executed only the first time
    - Cloning: one can duplicate any Eiffel object, which rules out the singleton pattern (will be fixed in the next version of Eiffel)
    - Creation procedure of singleton should not be exported, only to SHARED_SINGLETON. But now descendants of SHARED_SINGLETON may still create an instance. Solution: Make SHARED_SINGLETON frozen (i.e. no descendants anymore)
        - Advantages
            - Straightforward
            - Compilers can optimize
        - Weakness
            - Against open-closed principle



- Java/C++
    - Static features

7 Quality assurance and testing

7.1 Testing basics

- Software quality assurance (QA) is a set of policies and activities to
  - Define quality objectives
  - Help ensure that software products and processes meet these objectives
  - Assess to what extent they do
  - Improve them over time
- Software quality
  - Product quality (immediate)
    - Correctness
    - Robustness
    - Security
    - Ease of learning
    - Efficiency
  - Product quality (long-term)
    - Extendibility
    - Reusability
    - Portability
  - Process quality
    - Timeliness
    - Cost-effectiveness
    - Self-improvement
- Quality can be defined negatively: Quality is the absence of “deficiencies”.
- *Failures* result from *faults* that are caused by *mistakes*.
  - Failure: person’s age appears negative
  - Fault: code for computing age yields negative value if birthdate is in 20th century
  - Mistake: failed to account for dates beyond 20th century
- Definition of testing
  - To test a software system is to try to make it fail.
- The overall process of testing
  - Identify parts of the software to be tested
  - Identify interesting input values
  - Identify expected results (functional) and execution characteristics (non-functional)
  - Run the software on the input values
  - Compare results and execution characteristics to expectations
- Test definitions
  - Implementation under test (IUT): the software (and possibly hardware) elements to be tested
  - Test case: Precise specification of one execution intended to uncover a possible fault
    - Required state and environment of IUT

        - Inputs
    - Test run: One execution of a test case
    - Test suite: A collection of test cases
    - Expected results: Precise specification of what the test is expected to yield in the absence of a fault
        - Returned values
        - Messages
        - Exceptions
        - Resulting state of program and environment
        - Non-functional characteristics (time, memory)
    - Test oracle: A mechanism to determine whether a test run satisfies the expected results
        - Output is generally just “pass” or “fail”

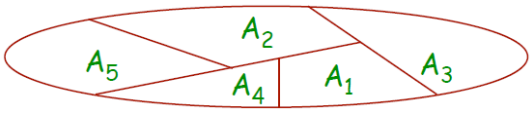
7.2 Classification of testing

- By scope
    - Unit test: tests a module
    - Integration test: tests a complete subsystem
    - System test: tests a complete, integrated application against requirements
- By intent
    - Functional testing
        - Goal: evaluate the system’s compliance with its specified requirements
    - Fault-directed testing
        - Goal: reveal faults through failures (unit and integration testing)
    - Conformance-directed testing
        - Goal: assess conformance to required capabilities (system testing)
    - Acceptance testing
        - Goal: enable customer to decide whether to accept a product
    - Regression testing
        - Goal: retest previously tested elements after changes to assess whether they have re-introduced faults or uncovered new ones
    - Mutation testing
        - Goal: introduce faults to assess test case quality
- Alpha and beta testing
    - Alpha testing: The first test of newly developed hardware or software in a laboratory setting
    - Beta testing: A test of new or revised hardware or software that is performed by users at their facilities under normal operation conditions
- By available information
    - White-box testing
        - To define test cases, source code of IUT is available
    - Black-box testing
        - Properties of IUT available only through specification

	White-box	Black-box
IUT internals	Knows internal structure & implementation	No knowledge
Focus	Ensure coverage of many execution possibilities	Test conformance to specification
Origin of test cases	Source code analysis	Specification
Typical use	Unit testing	Integration & system testing
Who?	Developer	Developers, testers, customers

7.3 Input partitioning

- Partition testing (black-box)
  - Impossible to test all inputs, but realistic inputs are desirable
  - Partition input set, i.e. a set of subsets that is
    - Complete: union of all subsets cover entire domain
    - Pairwise disjoint: no two subsets intersect



  - Purpose or hope
    - For any input value that produces a failure, some other in the same subset produces a similar failure
  - Ideas for equivalence classes
    - Values at the center of a range
    - Boundary values
    - Values known to be particularly relevant
    - Values that must trigger error messages
  - Partition testing is applicable at all levels of testing, i.e. unit, class, integration and system testing

7.4 Measure test quality

- To assess the effectiveness of a test suite, one could measure how much of the program is exercised

7.4.1 Coverage criteria

- Instruction (or statement) coverage
  - Measure instructions executed
  - Disadvantage: insensitive to some control structures
- Branch coverage
  - Measure conditionals whose paths are both executed
- Condition coverage
  - Count how many atomic Boolean expressions evaluate to both true and false

- Path coverage
    - Count how many of the possible paths are taken (a path is a sequence of branches from routine entry to exit)

7.4.2 Specification coverage

- Definitions
    - A **predicate** is an expression that evaluates to a Boolean value, e.g. $a \wedge b$
    - A **clause** is a predicate that does not contain any logical operator, e.g. $x < 1$
- Predicate coverage PC
    - A predicate is covered if and only if it evaluates to both true and false in 2 different runs of the system.
- Clause coverage CC
    - The clause coverage is satisfied if every clause of a certain predicate evaluates to both true and false.
- Combinatorial coverage CoC
    - Every combination of evaluations for the clauses in a predicated must be achieved.

7.4.3 Mutation testing

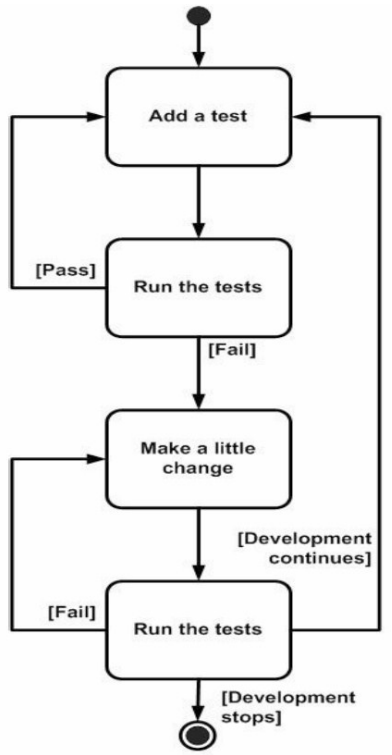
- Idea
    - Make small changes to the program source code and see if your test cases fail for the modified versions
- Purpose
    - Estimate the quality of your test suite
- Terminology
    - Faulty version of the program are called **mutants**
    - A mutant is said to be **killed** if at least one test case detects the fault injected into the mutant. Similarly, it is called **alive** if no test case detects the fault.
    - A **mutation operator** is a rule that specifies a syntactic variation of the program text so that the modified program still compiles. A mutant is then the result of an application of a mutation operator.
- Mutation operator coverage (MOC): For each mutation operator, create a mutant using that mutation operator.
- Classical mutation operators
    - Replace arithmetic/relational/logical operator by another
    - Replace a variable by another or a constant
    - Replace condition of a test by negation
    - Replace call to a routine by another
- OO mutation operators
    - Access modifier change (visibility)
    - Constructor call with child class type (polymorphism)
    - Argument order change

7.5 Unit testing

- Often done using xUnit framework, e.g. JUnit or CppUnit

7.6 Test-driven development (TDD)

- Software development methodology, one of the core practices of extreme programming (XP)
- Overview
  - Write a small test case
  - Write enough code to make the test succeed
  - Clean up the code
  - Repeat
- Properties
  - Evolutionary approach to develop
  - Combines test-first development and refactoring
  - Primarily a method of software design, not just a method of testing
- TDD 1: Test-first development (TFD)

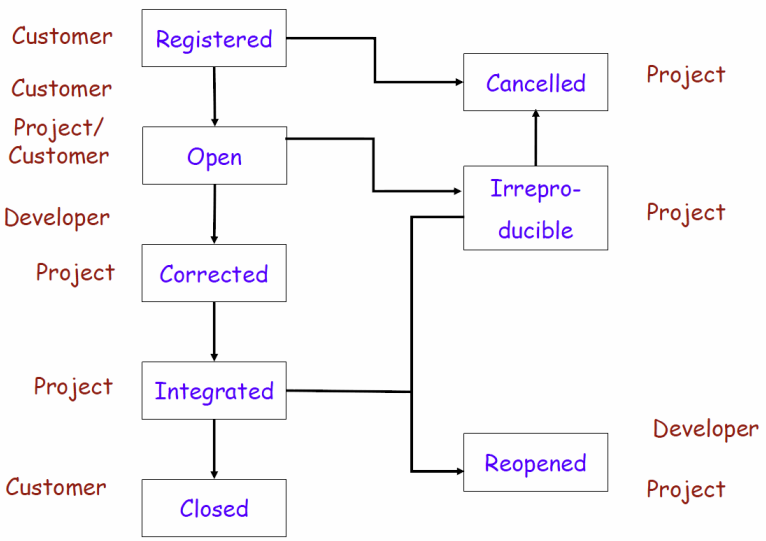


- TDD 2: Refactoring
  - A change to the system that leaves its behavior unchanged, but enhances some non-functional quality, like:
    - Simplicity
    - Understandability
    - Performance
  - Refactoring does not fix bugs or add new functionality
  - Examples

        - Change the name of a variable, class, ..
        - Generalize type
        - Break down large routine
- Documentation
    - Programmers often do not read documentation, instead they look for examples an play with them
    - Good unit tests can serve as examples and documentation
- Advantages
    - Reduce gap between decision and feedback
    - Encourage developers to write code that is easily tested
    - Creates a thorough test bed
- Drawbacks
    - Time taken away from core development
    - Some code is difficult to test

7.7 Test management

- Who tests?
    - Any significant project should have a separate QA team
- Classifying reports by severity
    - Not a fault,
    - Minor
    - Serious
    - Blocking
    - Enhancement
- Classifying reports by status
    - Registered
    - Open
    - Re-opened
    - Corrected
    - Integrated
    - Delivered
    - Closed
    - Irreproducible
    - Canceled



7.8 Debugging

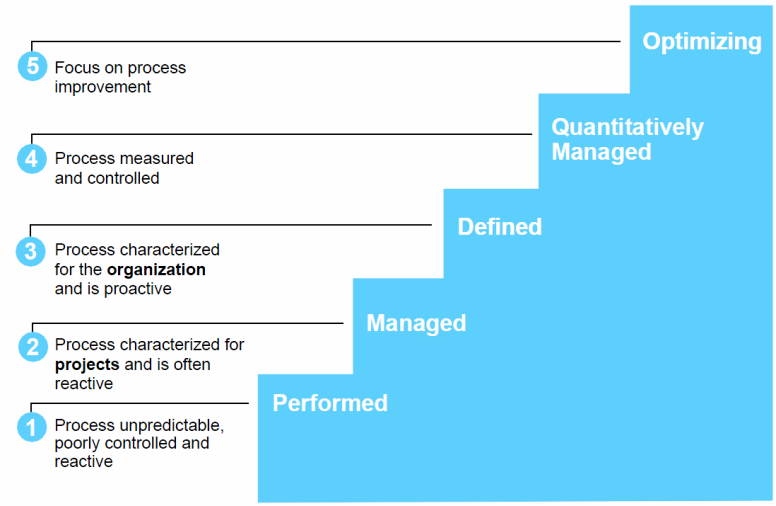
- Debugging is the work required to diagnose and correct a bug. That is, testing is neither debugging, nor is debugging testing.
- Delta-debugging
    - Simplification algorithm for bug reproducing examples
    - Reduces size of input or program
    - Easy to implement and customize

- Assumptions
  - Input can be split into parts
  - Working and failing program available
- Delta-debugging removes part of the input, and tries to find smaller inputs that still fail. This is done fully automatically
- Limitations
  - No guarantee for smallest possible input
  - Need to be able to split inputs
  - Empty input must not trigger failure

8 CMMI, PSP, TSP

8.1 CMMI

- Background
  - Initially Capability maturity model (CMM), developed for the US department of defense, meant for software
  - Generalized into CMMI
- The maturity levels
  - Capability levels are cumulative, levels build upon one another



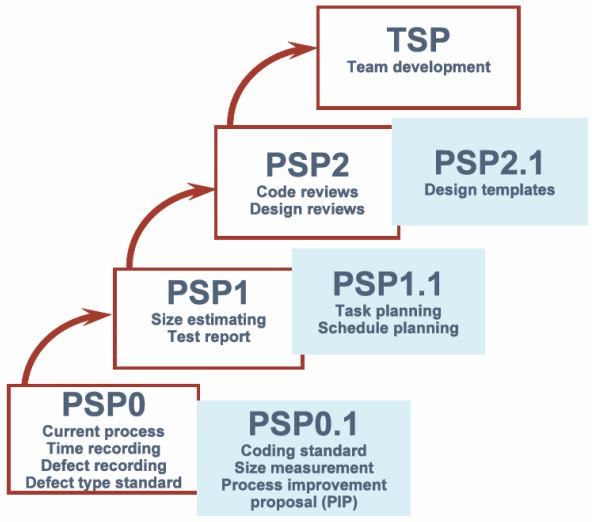
Level	Process Characteristics	Management Visibility
5 Optimizing	Focus is on continuous quantitative improvement	In Out
4 Quantitatively Managed	Process is measured and controlled	In Out
3 Defined	Process is characterized for the organization and is proactive	In Out
2 Managed	Process is characterized for projects and is often reactive	In Out
1 Initial	Process is unpredictable, poorly controlled, and reactive	In Out

- Basic ideas
  - Goal: determine the maturity level of the *process* of an organization, focusing on the process, not on technology
  - Emphasis *reproducibility* of results
  - Emphasis *measurement*, based on statistical quality control techniques

    - Relies on *assessment* by external team
- CMMI goals
    - Emphasis on developing processes and changing culture for measurable benefit to organization's business objectives
    - Framework from which to organize and prioritize engineering, people, and business activities
- CMMI covers
    - Systems engineering
    - Software engineering
    - Integrated product and process development
    - Supplier sourcing

8.2 PSP/TSP

- Transposition of CMMI-like ideas to work for individual teams and developers
- Initial objective: convince management to let the team be self-directed, meaning that it
    - Sets its own goals
    - Establishes its own roles
    - Decides on its development strategy
    - Defines its processes
    - Develops its plans
    - Measures, manages and controls its work
- Management will support you as long as you
    - Strive to meet their needs
    - Provide regular reports on your work
    - Do quality work
    - Respond to changing needs
    - To convince them of this, you must
        - Maintain and publish precise, accurate plans
        - Measure and track your work
        - Regularly show that you are doing a superior job
        - PSP helps you do this
- PSP essential practices
    - Measure, track and analyze your work
    - Learn from your performance variations
    - Incorporate lessons learned into your personal practices
- PSP fundamentals: As a personal process, PSP includes
    - Defined steps
    - Forms, standards
    - A measurement and analysis framework for characterizing and managing your personal work
    - A defined procedure to help improve your personal performance
-



- PSP0: Establish a measured performance baseline
    - PSP0: simple process, time/defect recording
    - PSP0.1: add size measurements
- PSP1: Make size, resource and schedule plans
    - PSP1: Establish size *estimation*
    - PSP1.1: Task and schedule planing
- PSP2: Practice defect and yield management
    - PSP2: Code/design review
    - PSP2.1: Design templates

9 Agile methods

- Process-oriented (sometimes called heavyweight or formal)
  - Examples
    - Waterfall model
    - CMMI
    - RUP
  - Overall idea is to enforce a strong engineering discipline on the software development process
    - Controllability, manageability
    - Traceability
    - Reproducibility
- Criticism on process-oriented
  - Requirements
    - Difficult to define in the beginning of a project
    - May change over time
    - May not capture what customer wants
  - Planning
    - Difficult because requirements change
  - Design and implementation
    - Reveals problems only late in the project
  - Testing
    - Reveals problems only at the end of a project
- Assembly-line versus prototype

Assembly-line manufacturing	Prototype-style manufacturing
Specify, then build	Hard to freeze specifications
Reliable effort and cost estimates are possible, early on	Estimates only become possible late, as empirical data emerge
Can identify schedule and order all activities	Activities emerge as part of the process
Stable environment	Many parameters change; need creative adaptation to change

  - In the agile view, most software development is not a predictable, mass-manufacturing problem, but falls under the new product development model
- History
  - Process-oriented was found to be
    - Heavy weight
    - Bureaucratic
    - Slow and demeaning
    - Inconsistent with how developers perform effective work
  - Software development seen as prototype-style manufacturing

    - Reaction: lightweight methods such as scrum, XP
    - In 2001, lightweight method became agile methods and the agile manifesto defined principles
- The agile manifesto
    - We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
        - Individuals and interactions over processes and tools
        - Working software over comprehensive documentation
        - Customer collaboration over contract negotiation
        - Responding to change over following a plan
- 13 agile principles
    - 1. Satisfy customers (early & continuous delivery)
    - 2. Embrace changing requirements
    - 3. Deliver working software frequently (weeks – months)
    - 4. Business people & developers collaborate daily
    - 5. Build projects around motivated individuals
    - 6. Face-to-face conversations
    - 7. Progress = working software
    - 8. Agile processes promote sustainable development
    - 9. Constant pace for all stakeholders
    - 10. Technical excellence and good design
    - 11. Goal is simplicity (maximize work not done)
    - 12. Self-organizing teams
    - 13. Reflection on team effectiveness and improvements

9.1 Basic concepts

- Iterative development
    - Time boxed iterative development: Set iteration end date, no change permitted
        - If requests cannot be met within timebox, place lower priority back on wish list
        - Iterations may typically last from 1 to 6 weeks
    - Parkinson's law: Work expands so as to fill the time available for its completion
- Requirements
    - Arguments against upfront requirements
        - Details are too complex
        - Stakeholders are not sure what they want and have difficulties stating it
    - The agile view:
        - Requirements always change
        - Developers never get complete specifications
- Customer on site
    - One or more customers sit full-time with the development team
        - Decisions on requirements and priorities
        - Explanations of features to the programmer

- User stories
    - Are a reminder to have a conversation with your stakeholders
    - Requirement formulated in everyday language
- Pair programming
    - All code is produced by two programmers at one computer
        - Rotate the input device
        - Pairs change frequently
        - Observer reviews code in real time
    - Benefits
        - Cross learning
        - Peer pressure
        - Help when programmer is stuck
- Test-driven development
- Self-organizing teams: development team organizes
    - How to fulfill the goals
    - How to solve problems
    - How to plan the work
    - What does the manager do then?
        - Coaching
        - Resources
        - Vision
        - Promotion of agile principles
    - Manager does not
        - Create a work breakdown structure, schedule or estimates
        - Tell people what to do

9.2 Scrum

- Practices
    - Self-directed and self-organized teams of max 7 people
    - No external addition of work to an iteration
    - Daily team measurement via a stand-up meeting called “scrum meeting”
    - 30 calendar-day iterations (“scrum sprints”)
    - Demo to stakeholders after each iteration
- Scrum lifecycle
    - Planning
        - Purpose
            - Establish the vision
            - Set expectation
            - Secure funding
        - Activities
            - Write vision

                - Write budget
                - Write initial product backlog
                - Estimate items
                - Exploratory design and prototypes
        - Staging
            - Purpose
                - Identify more requirements and prioritize enough for first iteration
            - Activities
                - Planning
                - Exploratory design and prototypes
        - Development
        - Release

9.3 XP

- Practices (about people)
    - Team typically works in an open space
    - Stakeholders are mostly available on site
    - Every developer chooses his task
    - Pair programming, TDD, continuous integration
    - No overtime
    - Short iterations
    - Documentation is reduced to a bare minimum (favors oral communication)
- Why extreme?
    - Testing is good: do it from day 1 (TDD)
    - Code reviews are good: do them instantly (pair programming)
    - Frequent integration is good: 24/7 on dedicated build machine
    - Short iterations are good: 1- 3 weeks
    - Customer involvement is good: bring customers on site
- Lifecycle
    - Exploration
        - Ensure feasibility
        - Prototypes
        - Story card writing and estimating
    - Planning
        - Release planning game
    - Iterations to first release
        - Implement a tested system ready for release
    - Productizing
        - Operational deployment
        - Documentation, training
    - Maintenance

        - Enhance, fix

9.4 RUP

- RUP is a process-based iterative approach
- Rational unified process
    - Iterative and incremental development process
    - Extensible framework to be customized
    - Based on spiral model
- Practices
    - Risk-driven requirements
    - Visual modeling (whiteboard)
    - Develop in short time-boxed iterations
- Lifecycle
    - Inception (days)
        - Requirements workshop, 10% of requirements captured
        - Goal: identify scope, vision, priorities, risks
    - Elaboration (iterations)
        - Core elements programmed and tested
        - Goal: stabilize vision, requirements and architecture
    - Construction (iterations)
        - Build remainder of the system, alpha testing, performance tuning, document creation
        - Goal: system ready for deployment
    - Transition (iterations)
        - Release candidate for feedback, distribution, education
        - Goal: system deployed
- Main intentions
    - Attack risks early and continuously
    - Deliver customer value early and often
    - First focus on software development, then on documentation
    - Work component-oriented: reuse

9.5 Criticism of XP

- Customer on site: difficult to find them
- Lack of documentation: difficult for maintenance
- Refactoring may introduce faults
- Hype not backed by evidence of success
- Distributed teams? Large projects/teams?

10 Distributed and outsourced software engineering (DOSE)

10.1 Motivations

- Money!
- Offshoring propositions
    - Low salaries
    - Skilled workforce
    - Good university system
    - Good communication infrastructure
    - Stable political structure
- Arguments for outsourcing
    - Cost (!)
    - Access to expertise
    - Focus on core business
    - Speed
    - Quality improvement
- Arguments against outsourcing
    - Loss of control, dependency on supplier
    - Loss of expertise
    - Loss of flexibility
    - Loss of jobs, effect on motivation

10.2 Challenges in DOSE

- Project management
    - Provide templates
    - Monitor tasks constantly
    - Maintain regular communication
    - Commit rules
    - Code reviews
- Cultural differences
    - Different cultural backgrounds, national holidays and interpretations
- Time zones
    - Keep meetings on schedule
- Communication and language skills
    - Email is not enough, need for voice communication
    - Heavy accents
    - Use several forms of communication, like email, voice, wikis
    - Send important information in writing

11 Modeling with UML

11.1 Introduction

- What is modeling
  - Building an abstraction of reality
    - Abstractions from things, people, processes
    - Relationships between these abstractions
  - Abstractions are simplifications
    - They ignore irrelevant details
    - They represent only the relevant details
    - What is relevant depends on the purpose of the model
  - Draw complicated conclusions in the reality with simple steps in the model
- Why model software?
  - Software is getting increasingly more complex
  - Code is not easily understandable by developers who did not write it
  - Need for simpler representations

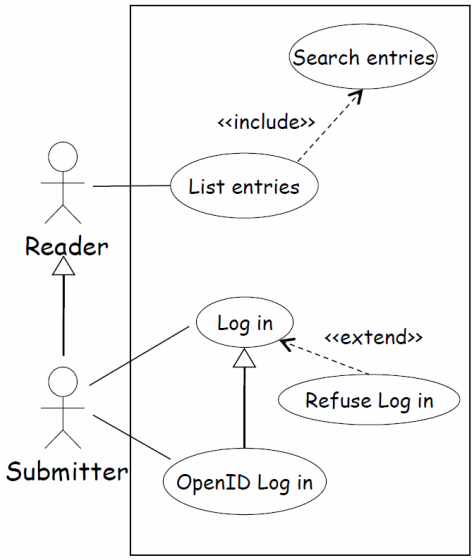
11.2 UML introduction

- Unified modeling language
- Uses of UML
  - Specification: The language is supposed to be simple enough to be understood by clients
  - Visualization: Models can be represented graphically
  - Design: the language is supposed to be precise enough to make code generation possible
  - Documentation: the language is supposed to be widespread enough to make your models understandable by other developers

11.3 Canonical diagrams

- Functional
  - Use case diagrams (requirements, client's point of view)
- Static structure
  - Class diagram (classes and their relationship)
  - Object diagram (relationship between objects at an interesting point in time)
  - Composite structure diagram (internal structure of a class)
  - Package diagram (packages and relationship between them)
  - Implementation diagram
    - Component diagram (physical components and relationship)
    - Deployment diagram (assigning components to nodes)
- Behavioral
  - State diagram (object lifecycle)
  - Activity diagram (flowchart, algorithm description)
  - Interaction diagrams

- Sequence diagram (message passing, ordered in time)
- Communication diagram (message passing)
- Interaction overview diagram (activity diagram with interaction diagrams in nodes
- Timing diagram (focus on timing constraints)



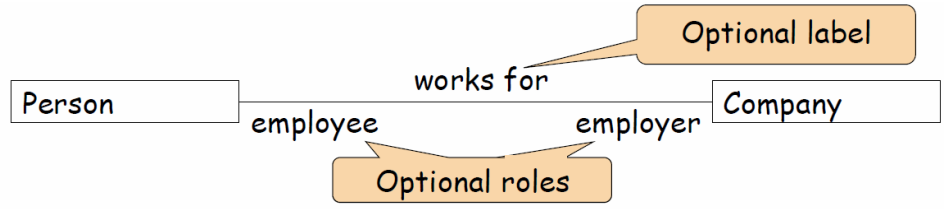
11.4 Diagram types

11.4.1 Use case

- Elements
  - Unique name
  - Participating actors
  - Entry condition
  - Flow of events
  - Exit condition
  - Special requirements

11.4.2 Classes

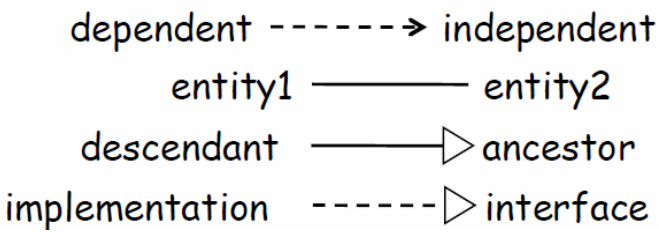
- Elements
  - Name (mandatory)
  - Attributes with type
  - Operations with signature
- Associations
  - Most widely used relation on class diagrams
  - In general means that classes know about each other, their objects can send each other messages (call operations, read attributes)
  - Special cases
    - Class A has an attribute of type B
    - Class A creates an instance of B
    - Class A receives a message with argument of type B
  - Mostly binary, but can be n-ary



- Aggregation
  - Part of relation between objects, where an object can be part of multiple objects
  - Part can be created and destroyed independently of aggregate

- Composition
  - Strong aggregation
  - An objects can only be part of a single other object, and exists only together with aggregate

- Relations
  - Dependency – changing the independent entity may influence the dependent one
  - Association – entities are directly connected (e.g. aggregation)



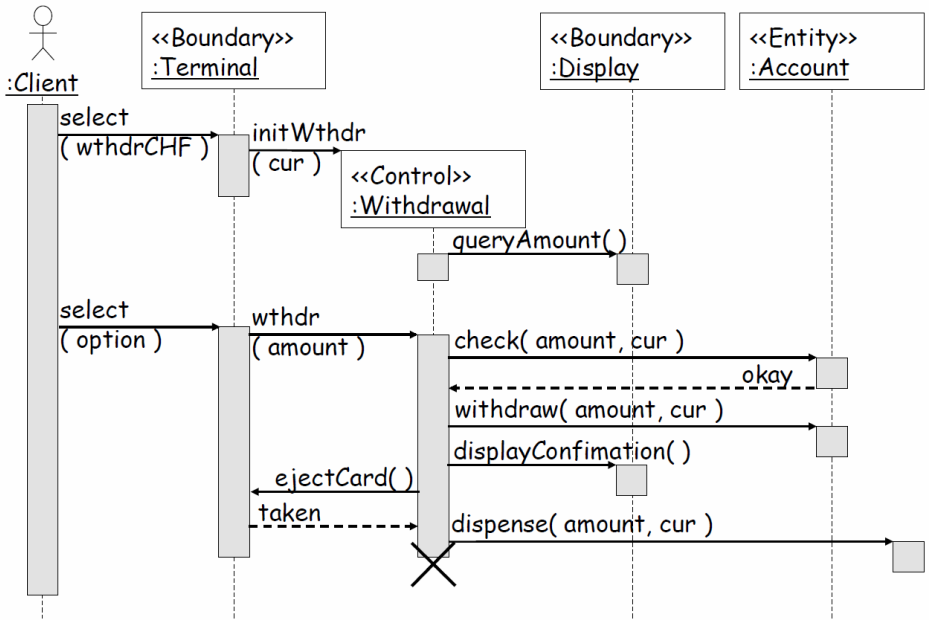
  - Generalization – an entity is a special case of another
  - Implementation – an entity is an implementation of another (e.g. interface)

11.4.3 Dynamic model

- We look for objects that are interaction and extract their "protocol" => sequence diagrams
- We look for objects that have interesting behavior of their own => state diagrams

11.4.3.1 Sequence diagrams

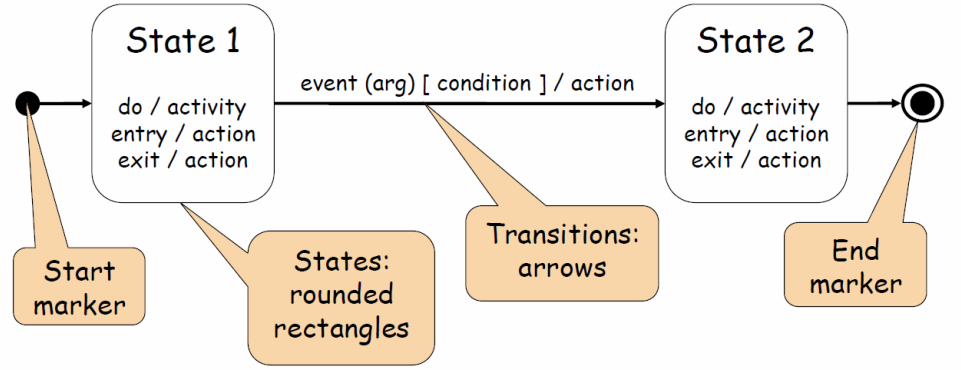
- Entities: objects
- Relations: message passing
- Sugar: lifelines, activations, creations, destructions, frames



11.4.3.2 State diagrams

- Objects with extended lifespan often have state-dependent behavior

    - Typical for control objects
    - Less often for entity objects
    - Almost never for boundary objects
- Event
    - Something that happens at a point in time, e.g. receipt of message
- Action
    - Operation in response to an action
- Activity
    - Operation performed as long as object is in some state

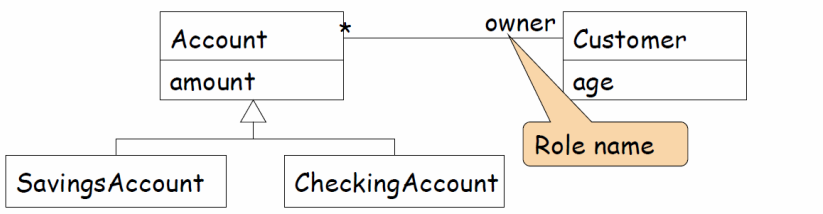


11.5 Object constraint language

- UML is not enough, for instance it is not possible to express that the spouse of a spouse should be the person itself
- These contracts can be expressed by different means
    - Natural language
        - Easy to understand and use, but ambiguous
    - Mathematical notation
        - Precise, but difficult for normal customers
    - Contract languages like Eiffel or JML
        - Formal, but easy to use

11.5.1 Object constraint language OCL

- Contract language for UML
- Used to specify invariants of objects, pre- and post conditions of operations and guards (e.g. in state diagrams)
- Constraints can mention
    - Self: the contextual instance
    - Attribute and role names
    - Side-effect free methods (stereotype <<query>>)
    - Logical connectives
    - Etc



A savings account has a non-negative balance

```
context SavingsAccount inv:
self.amount >= 0
```

Checking accounts are owned by adults

```
context CheckingAccount inv:
self.owner.age >= 18
```

12 Designing for concurrency – the SCOOP approach

- SCOOP introduces an extension to the type system: Variables might be *separate*.
- Fundamental semantic rule: x.r(a) waits for non-separate x, and doesn't wait for separate x.
- Wait by necessity
    - No explicit mechanism needed for client to resynchronize with supplier after separate call. The client will wait only when it needs to, i.e. when a query is performed.

12.1 Mutual exclusion

- Separate argument rule
    - Target of a separate call must be formal argument of enclosing routine.
- Wait rule
    - A routine call with separate arguments will execute when all corresponding processors are available and hold them exclusively for the duration of the routine
- Condition synchronization
    - A call with separate arguments waits until
        - All corresponding objects are available
        - Preconditions hold
    - i.e. preconditions become wait conditions, no need for notification

13 Architectural styles

- An architectural style is defined by
  - Type of basic architectural components (like classes, filters, databases, layers)
  - Type of connectors (calls, pipes, inheritance)

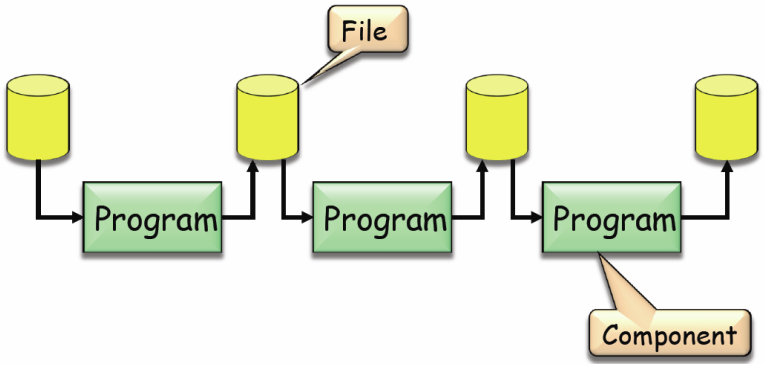
13.1 Examples

13.1.1 Concurrent processing

- Strengths
  - Separation of concerns
  - Performance
  - Provide user with ability to perform several task at once (e.g. browser tabs)
- Weaknesses
  - Difficulty of synchronization
  - Must find out what parallelizable
  - Limits to performance improvements (Amdahl's law)
    - $speedup = \frac{1}{(1-p)+p/n}$

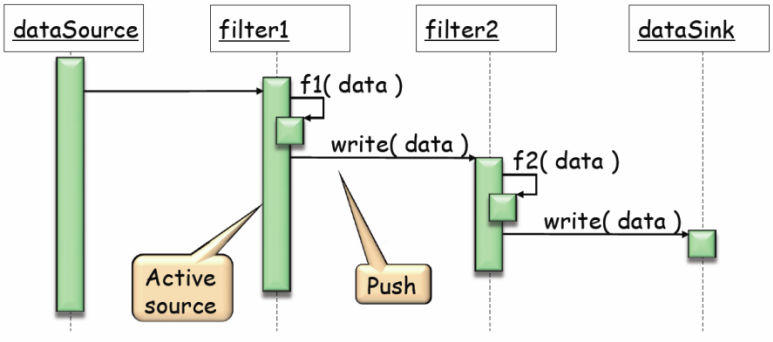
13.1.2 Dataflow systems

- Availability of data controls computation. The structure is determined by the orderly motion of data from component to component
- Batch-sequential
  - Frequent architecture in scientific computing and business data processing
  - Components are independent programs, connectors are media (typically files)



- Pipe-and-filter
  - Components (filters)
    - Read input stream
    - Locally transform data
    - Produce output streams
  - Connectors (pipes)
    - Streams, e.g. FIFO buffers
  - Data is processed incrementally, as it arrives
  - Filters must be independent

        - Filters do not share state
        - Filters do not know upstream and downstream filters



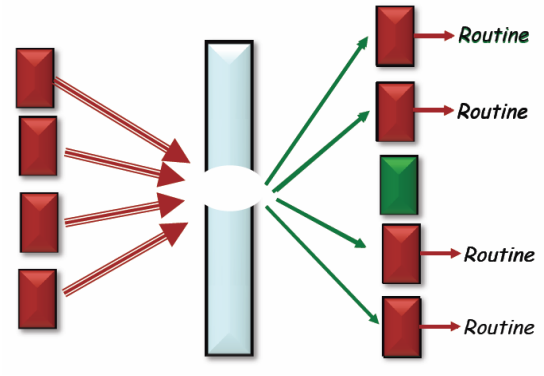
    - Strengths
        - Reuse filters
        - Ease of maintenance: filters can be added and replaced
        - Potential for parallelism, because filters are independent
    - Weaknesses
        - Sharing global data expensive or limited
        - Scheme highly dependant on order of filters
        - Can be difficult to design incremental filters
        - Not appropriate for interactive applications
        - Error handling is difficult
        - Data type must be greatest common denominator

13.1.3 Call and return

- Functional
    - Components are routines and connectors are routine calls
    - Strengths
        - Change implementation without affecting clients
        - Reuse of individual operations
    - Weaknesses
        - Must know which exact routine to change
        - Hide role of data structure
        - Bad support for extendibility
- Object-oriented
    - Components are classes and connectors are again routine calls
    - Strengths
        - Change implementation without affecting clients
        - Break problems into interacting agents
    - Weaknesses
        - Objects must know their interaction partners
        - Side effects

13.1.4 Event-based

- Publish-subscribe

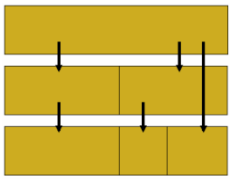


- Strengths
  - Strong support for reuse
  - Maintenance: add and replace components with minimum effect on system
- Weaknesses
  - Loss of control (what components responds to an event?)
  - Correctness hard to ensure

13.1.5 Data centered

- Components are central data store that represents state
- Interdependent components operate on data store
- Blackboard architecture (special case)
  - Interactions among knowledge source solely through repository
- Strengths
  - Efficient way to share large amounts of data
  - Data integrity localized to repository
- Weaknesses
  - Subsystems must agree (i.e. compromise) on a repository data model
  - Distribution can be a problem

13.1.6 Hierarchical (layered)



- Components are group of subtasks which implement an abstraction at some layer
- Connectors are the protocols that define how layers interact
- Strengths
  - Increasing levels of abstraction as we move up through layers
  - Maintenance: in theory, a layer only interacts with layer below
  - Reuse: different implementations of the same level can be interchanged
- Weaknesses
  - Performance: communication down through layers and back up
- Interpreters
  - Architecture based on a virtual machine produced in software

    - Special kind of layered architecture
    - Components are the program being executed and its data

13.1.7 Client-server

- Components are subsystems that are independent processes. Servers provide a specific service, and clients use these services
- Connectors are data streams, typically over a communication network
- Strengths
    - Makes effective use of networked systems
    - May allow for cheaper hardware
    - Easy to add new servers
    - Availability through redundancy
- Weaknesses
    - Data interchange can be hampered
    - Communication might be expensive
    - Single point of failure

13.1.8 Peer-to-peer

- Similar to client-server, but each component is both server and client
- Strengths
    - Efficiency: all clients provide resources
    - Scalability
    - Robustness as data is replicated, i.e. no single point of failure
- Weaknesses
    - Architectural complexity
    - Resources are distributed and not always available
    - More demanding of peers

14 Measurement

14.1 Introduction

- Why measure software?
  - Understand issues of software development
  - Make decisions on basis of facts rather than opinions
  - Predict conditions of future development
- Methodological guidelines
  - Measure only for a clearly stated purpose
  - Specifically, software measures should be connected with either quality or cost (or both)
  - Assess the validity of measures through controlled, credible experiments
  - Apply software measures to software, not people
- What to measure
  - Effort measures
    - Development time
    - Team size
    - Cost
  - Quality measures
    - Number of failures
    - Number of faults
    - Mean time between failures

14.2 Metrics

- Traditional internal code metrics
  - Source lines of code
    - Pros
      - Appeals to programs
      - Easy to measure
      - Correlates well with other effort measures
    - Cons
      - Ambiguous (several instructions per line, count comments?)
      - Does not distinguish between programming languages
      - Low-level, implementation-oriented
      - Difficult to estimate in advance
  - Comment percentage
  - McCabe cyclomatic complexity
    - A measure based on a connected graph of the module
    - $M = E - N + P$
      - M is the cyclomatic complexity
      - E is the number of edges in the graph
      - N is the number of nodes in the graph

        - P is the number of connected components in the graph
- External metrics
    - Function points
        - Input, output, inquiries, files and interfaces to other systems
    - Application points
        - High level effort generators, e.g. screens, reports, high-level modules

14.3 Cost models

- Purpose
    - Estimate in advance the effort attributes (development time, team size, cost) of a project
- Problems
    - Find the appropriate parameters defining the project
    - Measure these parameters
    - Deduce effort attributes through appropriate mathematical formulas

14.3.1 COCOMO

- Basic formula
    - $Effort = A \cdot Size^B \cdot M$
    - M is the cost driver estimation
    - A is a constant, e.g. 2.94
    - B is a constant depending on the novelty, risk, process
- For size, use:
    - Action points at state 1 (requirements)
    - Function points at state 2 (early design)
    - Function points and SLOC at stage 3 (post-architecture)

14.4 Reliability models

- Goal
    - Estimate the reliability – essentially the likelihood of faults – in a system
- Mean time to failure MTTF
- Mean time to repair MTTR
- Reliability R

$$R = \frac{MTTF}{1 + MTTF}$$

14.5 Goal/Quality/Metric GQM

- Process for a measurement campaign
    - Define goal of measurement
    - Devise suitable set of questions
    - Associate metric with every question