SOFTWARE REUSE
Architecture, Process and Organization for Business Success

Jacobson, Griss, Jonsson
Addison-Wesley, 1997

Lecture Slides
to Accompany the Text

Software Reuse (part 1)

- Beating the competition
  - » Faster:
    - software must meet market window set by competitive organizations
  - » Better:
    - software must serve requirements of the process it supports and with few failures
  - » Cheaper:
    - software must be less expensive to produce and maintain

Slides adapted from Software Reuse, Ivar Jacobson, Griss, Jonsson, Addison-Wesley, 1997

Software Reuse What and Why?

- develop systems of components of reasonable size and reuse them
  - » minimize redundant work
    - - environment / problem description
    - - unit and system testing
  - » "Passive Reuse" may result in 15-20% reuse
  - » "Systematic Reuse"
    - - Hitachi's Eagle : 60-98% reuse
    - - Hewlett-Packard : 25-50% reuse of firmware
    - - AT&T : 40-92% in Telecoms support software
    - - Brooklyn Union Gas : 90-95% in process layer and 67% in user interface and b-objects
    - - Ericsson AXE : 90 %
    - - Motorola 85% reuse and 10:1 in compiler tools

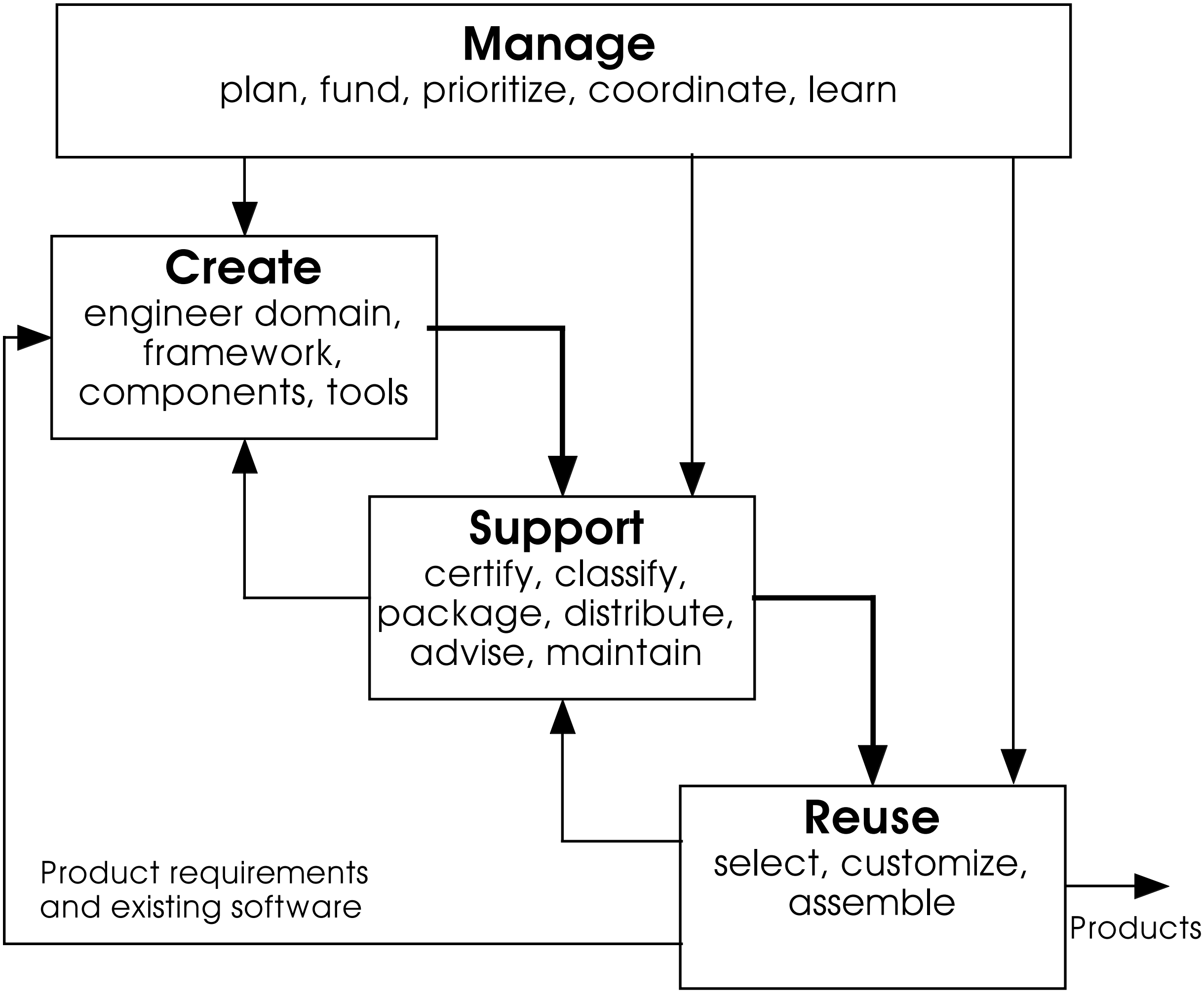
Software Reuse What happens?

- Time to market
  - » reduced 50 - 80%
- Defects
  - » reduced 80 - 90%
- Maintenance cost
  - » reduced 80 - 90%
- Software life cycle costs
  - » reduced 15-75%
- Product Quality
  - » highly customizable products
  - » increased market agility
  - » consistent families of related products
  - » familiar compatible interfaces

Software Reuse Challenges?

- Software engineering approaches to requirements, architecture, analysis, design, test, and implementation of clearly identified elements for reuse
- No effective component inventory: lack of packaging, documentation, cataloging, inadequate libraries
- Component inflexibility: host, architecture, language incompatibilities
- Lack of reuse-oriented support tools and environments
- Need for business models that support capital intensive, domain engineering

Systematic Software Reuse



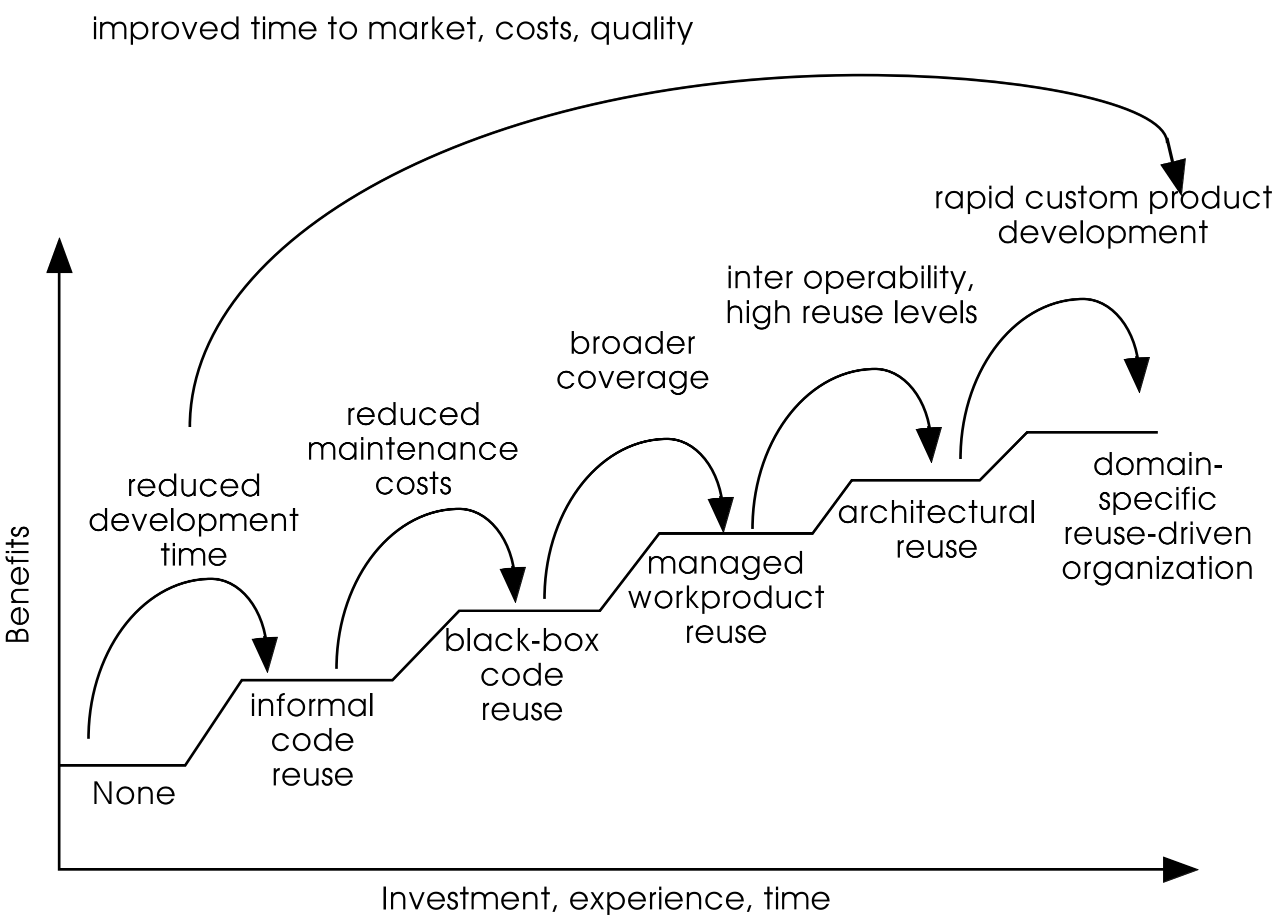
Systematic Software Reuse:

- **Create**: identify & produce
- **Reuse**: select, customize, assemble
- **Support**: certify, package, maintain
- **Manage**: plan, coordinate, measure
- **Domain Engineering**: identifying core structures and patterns that are shared by a family of application requirements within an application domain; resulting in customizable, configurable "parts" of applications in the domain
- **Application System Engineering**: specialization and assemble of domain "parts" to meet specific requirements

Organization:

- Classic software development organization focuses on creating application solutions
- Systematic Software Reuse organization must balance application solution creation with reusable asset creation and component stewardship
- Reuse must be “championed” organizationally with reuse as a “strategic” concern of uppermost management

Adopting Reuse Incrementally



Keeping the Faith!

» sustained top management leadership

» foster a system architecture, organization

» engender reuse as core to architecture

» manage create and use separately

» create components in the “real” world

» manage systems/components as assets

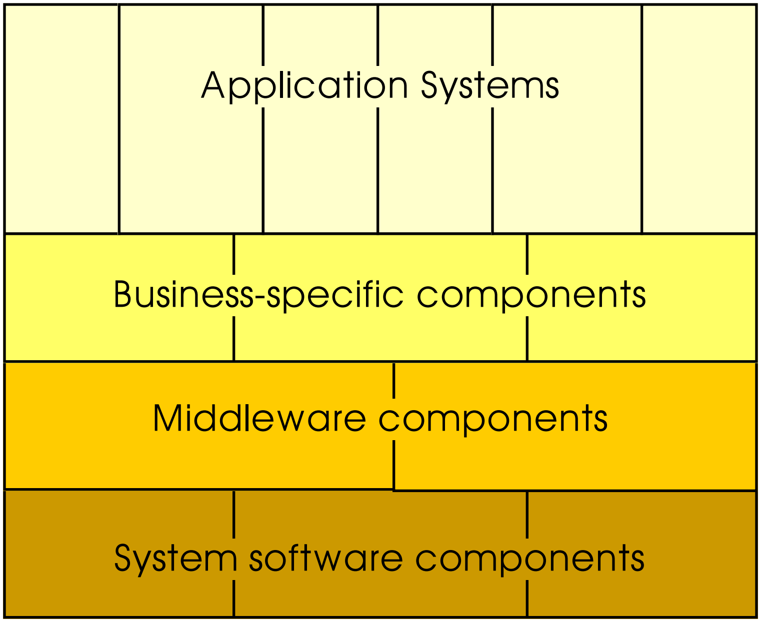
» technology and tools are not sufficient

» support champions and change agents

» invest in infrastructure and reuse education

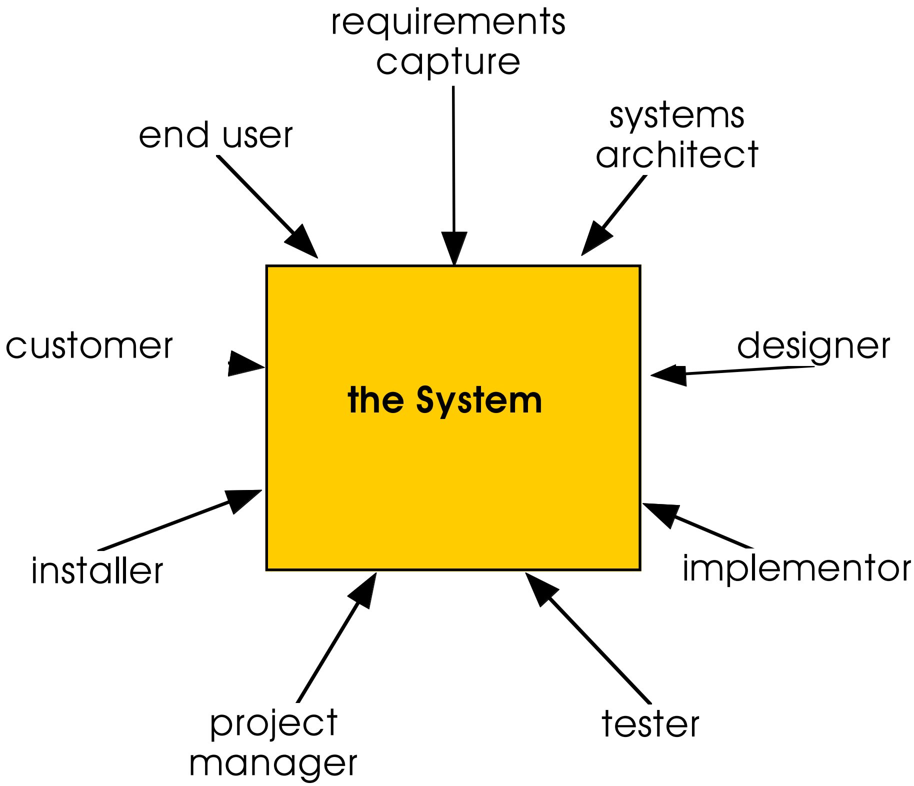
» measure with metrics and optimize to them

Architectural Style
(system layering)

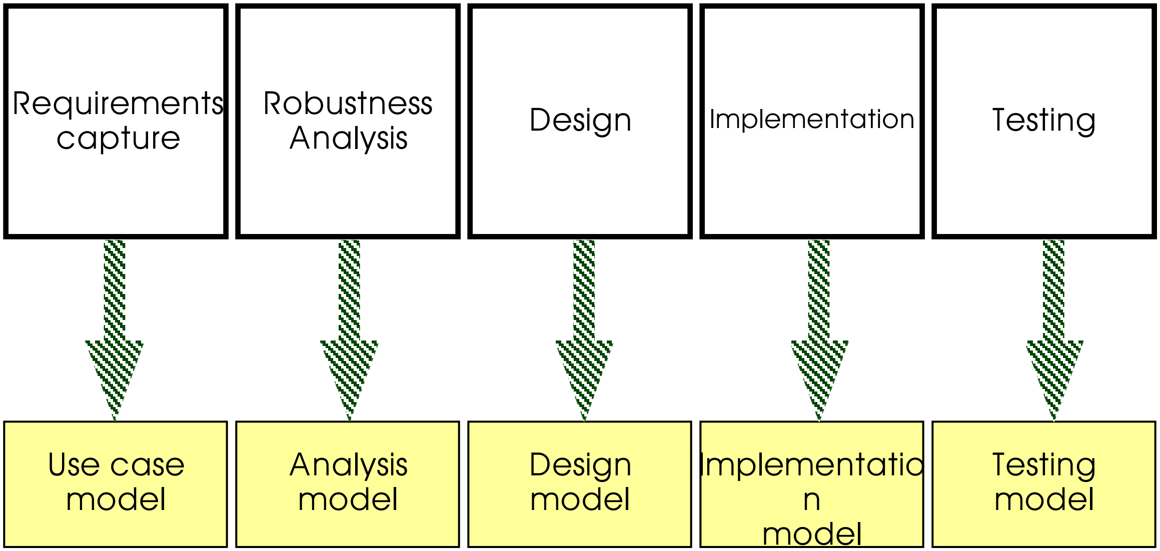


The Architectural style of a system is the denotation of the modeling languages used when modeling that system. *(Jacobson et al., 1992)*.

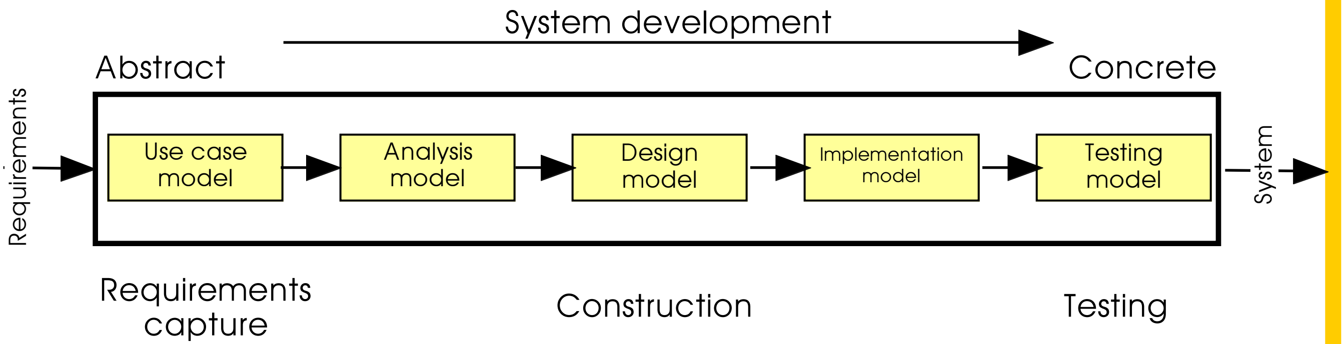
Object-Oriented Software Engineering



Main Activities of an SDLC



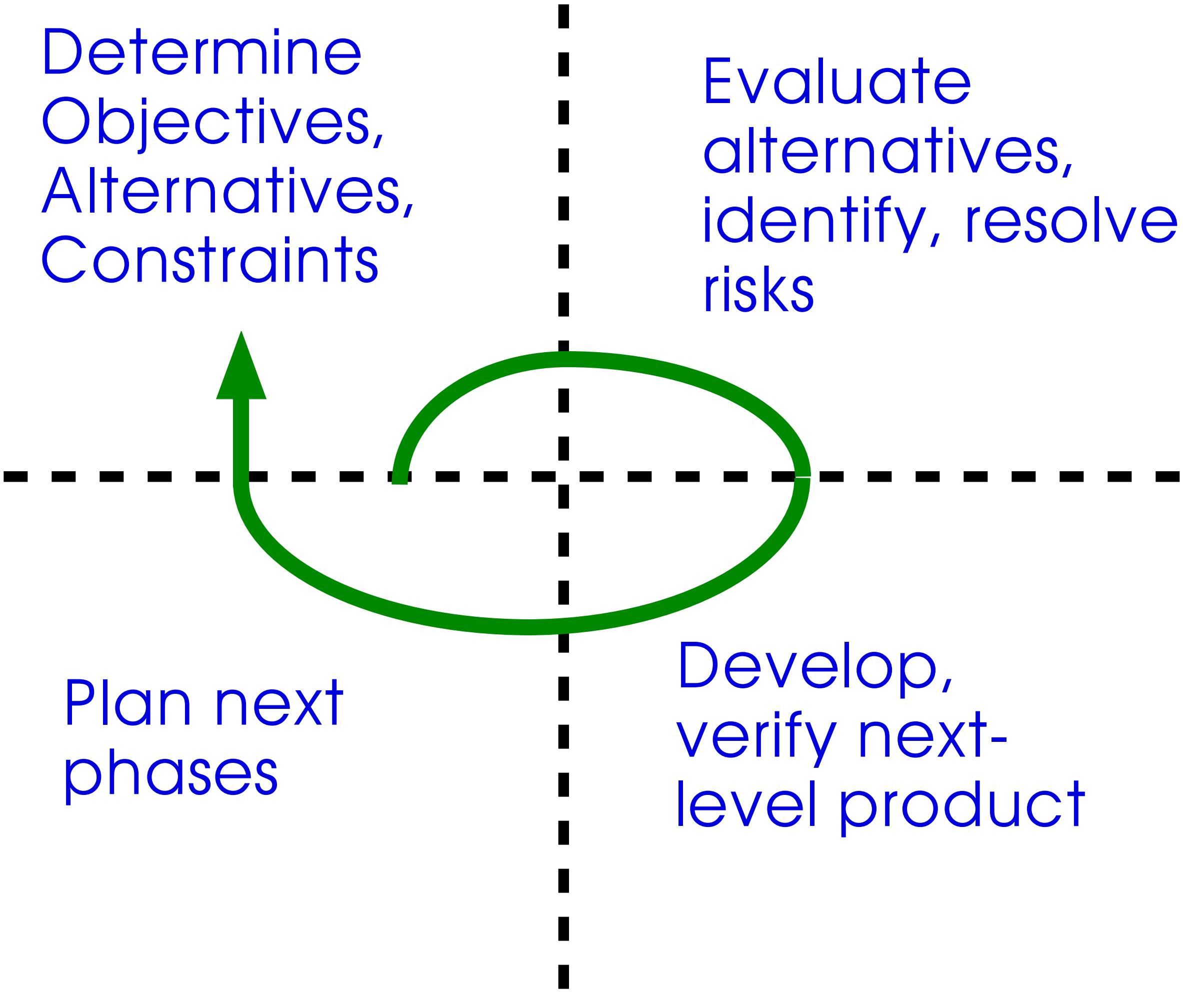
Software Engineering is systematic model building



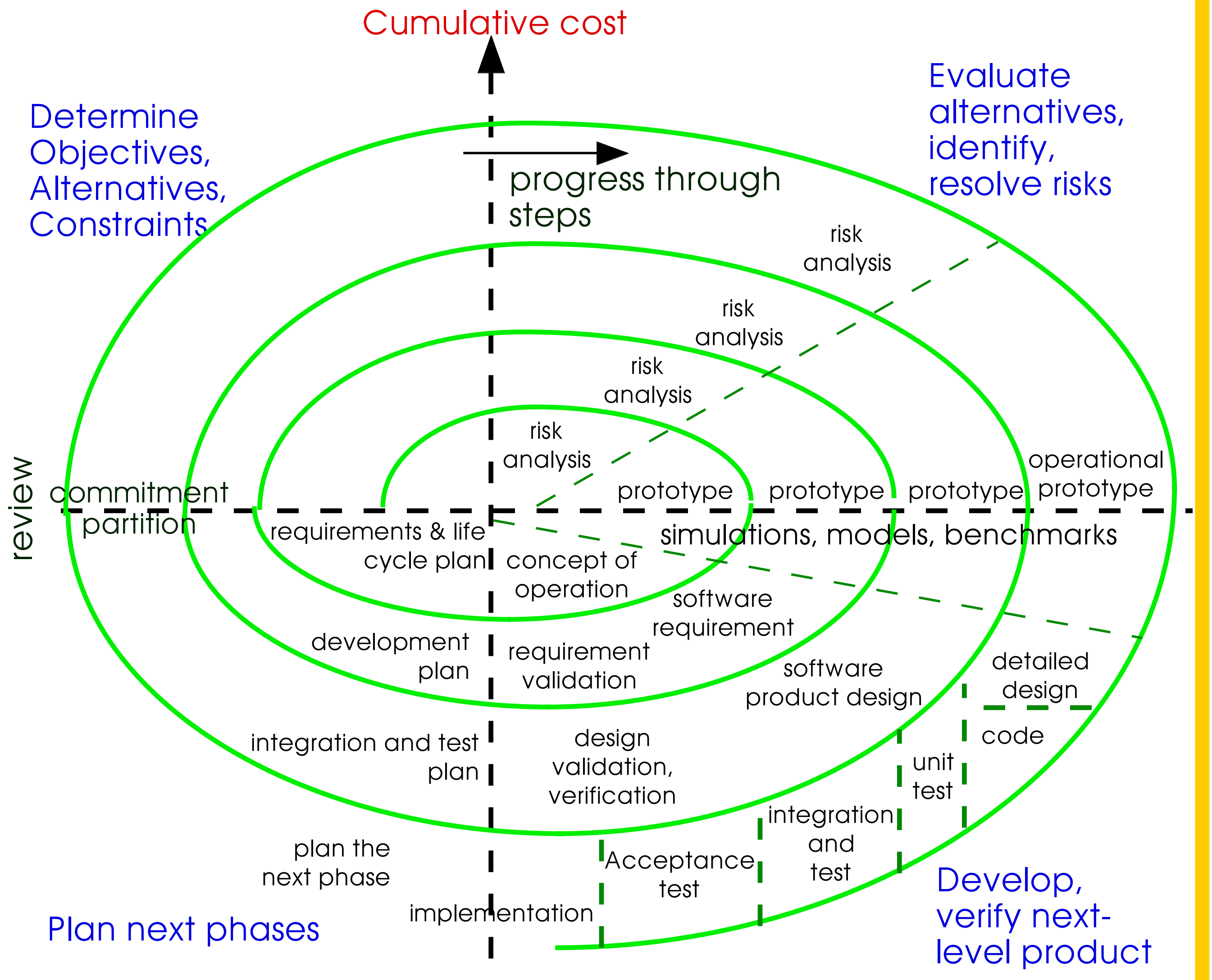
A Hybrid or Meta-Model Approach

- Spiral Life Cycle
  - expands the scope of cycle focus to process decisions as well as product decisions
  - focuses on risk analysis to guide process
  - revisits objectives, alternatives, constraints frequently
  - shapes subsequent cycle phases as part of the life cycle process
- It redefines the life cycle question
  - by subsuming the life cycle as a product in itself
  - allows other life cycle models to be special cases

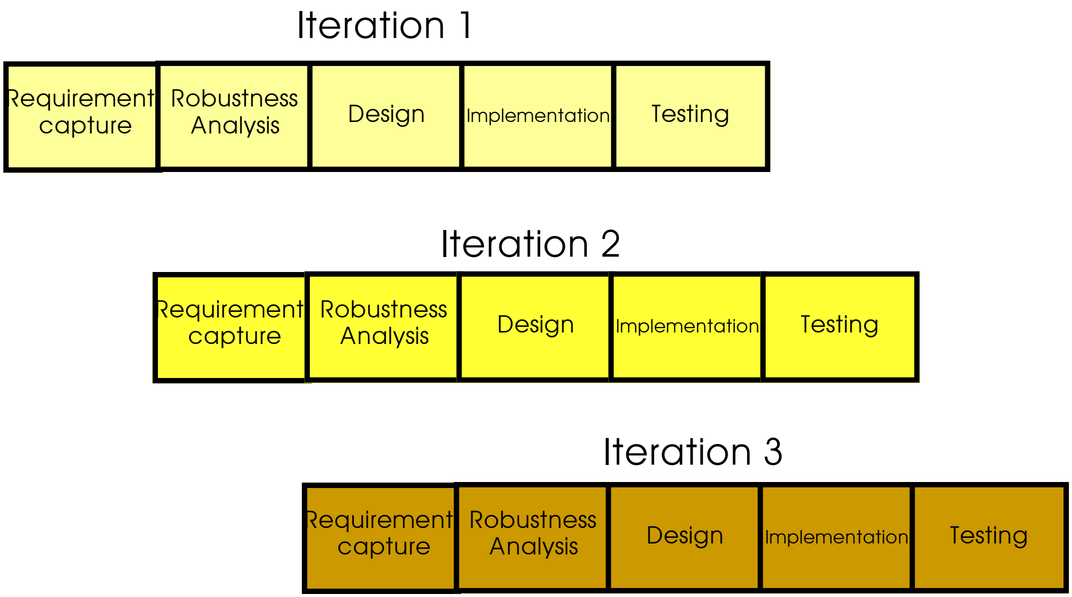
Spiral Model



Spiral Model



Incremental, Iterative OOSE Model Building



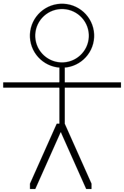
Objects unify the modeling process.

- **Objects** contain both behavior and data
- **UML** allows extensions called "**Stereotypes**" that can be used to define any modeling artifact, style or relationship needed (i.e. «boundary», «interface», «uses», «extends», …)
- **Packages** and subsystems collect classes, types and other elements for organization

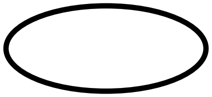
USE Cases capture system requirements

Requirements capture

- USE Cases define the “actors” and the “actions” that characterize the system responsibilities exhibited in an encounter with the system



» An **actor** is anything that interacts (exchanges data and events) with the system.



» A **use case** is a sequence of transactions performed by a system which yields and observable result of value for an actor.

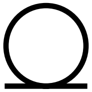
» **«uses»** defines the generalization of use case behavior inherited by a child case

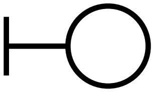
» **«extends»** describes a derived (or alternate) version of a use case

The analysis model shapes system architecture

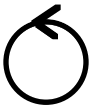
Robustness Analysis

- Architecture deals with principles, mechanisms, patterns, and structures that clearly communicate the structure of the system.
- Analysis models deal with the "ideal" implementation independent system functions
  - » **«entity»** objects depict long lived objects

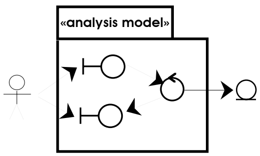




  - » **«boundary»** objects depict links between the system and environment, communicating



  - » **«control»** objects depict use-case-specific behavior



  - » **«analysis model»** is a stereotypical package collecting the analysis content

The design model defines the implementation

Design

- The design model is a “blueprint” for the system programming, how it is organized
- *Design classes* are more detailed than analysis classes, but are not “source code” level yet
- All analysis classes are mapped to one or more design classes (or fewer) based on the class and component libraries of the target platform
- This mapping is the “trace” that connects the models and results in “traceability”

The implementation model is the code

Implementation

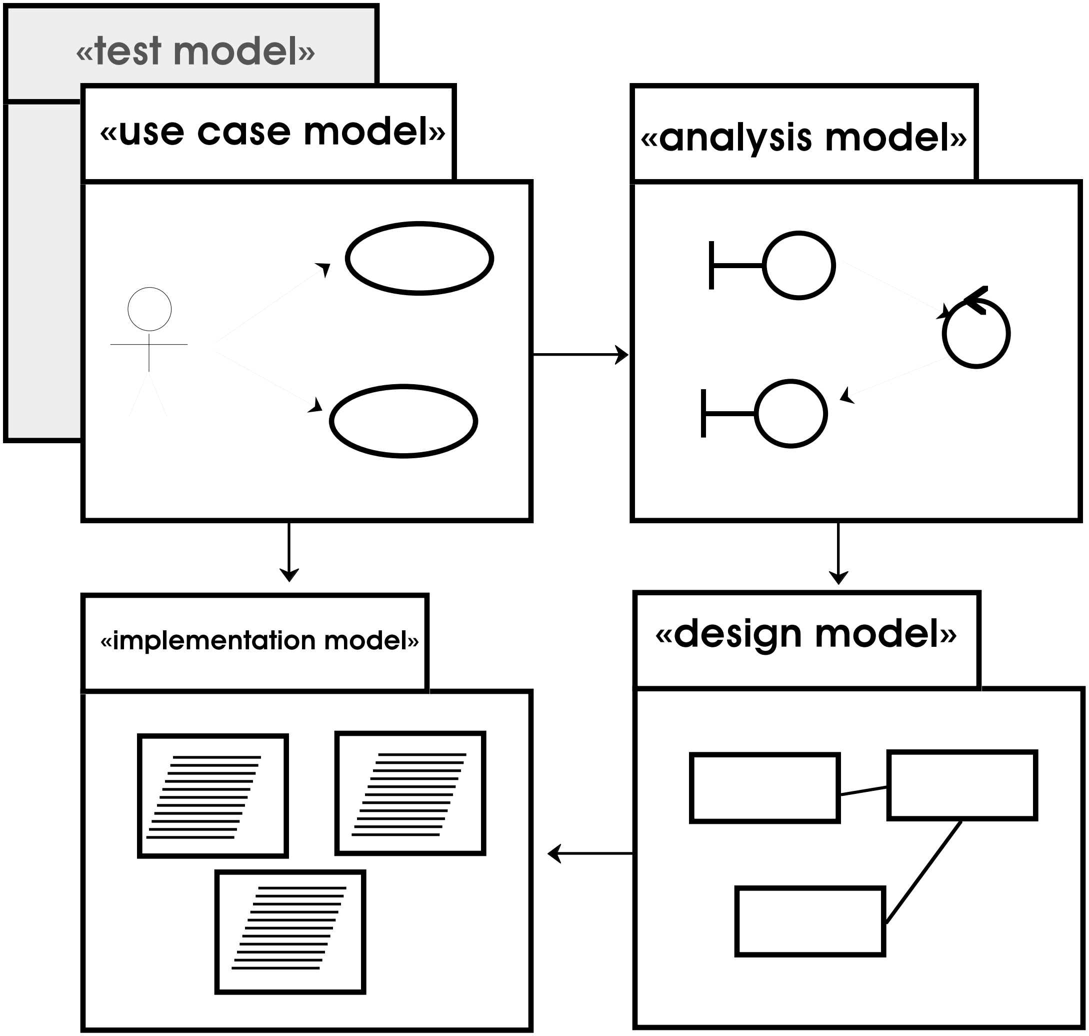
- Traced directly from the design model, the source code implements the relationships and behaviors defined in the design model
- In, particular method bodies are specific to the target programming platform, C++, Java, Smalltalk
- Specific interfaces to elements of a component library are defined in specific syntax (e.g. OMG's Interface Definition Language, IDL)

The test model validates the system

Testing

- The is the specification of all tests and their expected results
- A test is derived from a use case instance and attempts to exercise all core and extended use case behavior
- Test case development can proceed in parallel with analysis, design and implementation since tests are derived from the requirements documentation

OOSE System is a set of models



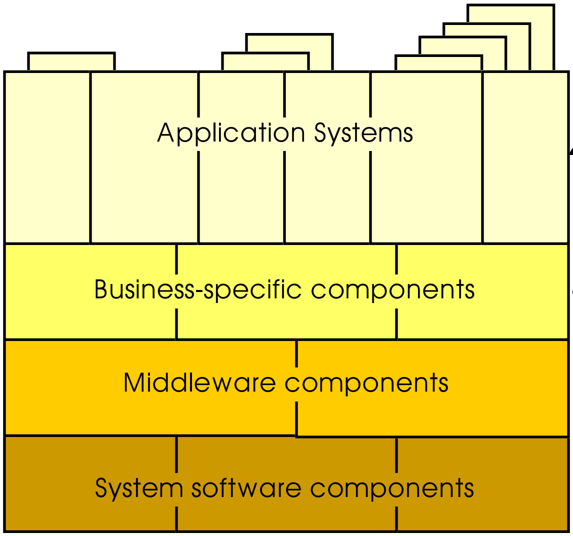
OOSE component reuse

- **application system**: is a system product delivered outside of the Reuse Business; when installed, it offers a coherent set of use cases to an end user
- **application system family**: is a set of application systems with common features
- **component**: is a type, class or any other workproduct that has been specifically engineered to be reusable
- **component system**: a system product that offers a set of reusable features, interrelated
- **facade**: a packaged subset of components providing access to a select set of component system features
- **variation point**: identifies a point of "inflection" in a reusable component

Variability mechanisms

- **Inheritance**: abstract methods, method overriding; sub-typing
- **Uses**: reusing an abstract use case concretely *«uses»*
- **Extensions and extension points**: attached variations in the normal flow of behavior, *«extends»*
- **Parameterizaton**: attribute variation (bounds...)
- **Configuration and module-interconnection languages**: selective inclusion of method bodies or implementations
- **Generation:** "macro" or "compiler-like" generation of source code based on selection/specification; sometimes "table-driven"

Layered Architecture
(virtual machine abstraction support)



Variants of a particular application system

Distinct application system - each application system offers a coherent set of use cases to some end users

Application domain and organization specific - contains a number of component systems specific to the type of business

Platform-independent distributed object computing - offers component systems for utility classes and platform-independent services for things like distributed object computing in heterogeneous environments

Platform-specific - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, and so on.