Software Development Processes

Steven Zeil

September 4, 2013

Contents

The workflows are important, but they are too low-level to guide us through the entire lifetime of a development project. As we saw in the earlier examples, the ADIV cycle can apply to a wide variety of different activities. How do we know *when* to move from class discovery to ADT design, and when to start coding?

We need some sort of overall strategic plan to help us choose appropriate topics of interest for our ADIV cycles. That's the purpose of a software development process.

Software Process Models

A *software development process* is a structured series of activities that comprise the way in which an organization develops software projects.

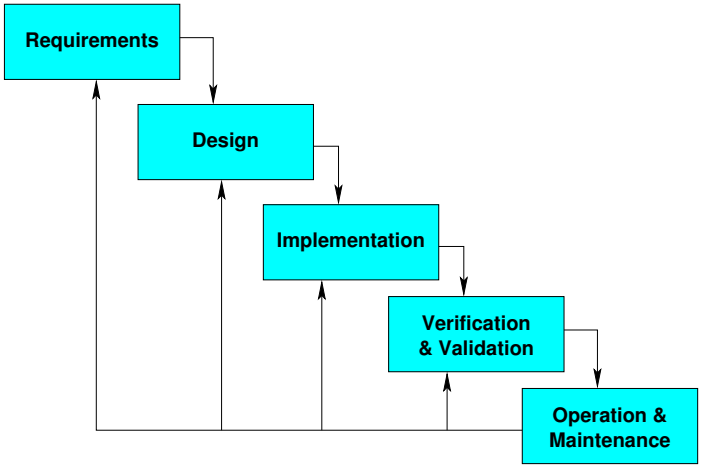
. . . . . .

1 The Waterfall Model

The Waterfall Model

Perhaps the best-known and most widely used process model is the *waterfall model*.

- Some of the names of the phases in this process model are the same as the names of some of our workflows.
- You need to rely on context to know whether "Design" is referring to a workflow or to a phase of the development process.



The waterfall model gets its name from the fact that the first diagrams of this process illustrated it as a series of terraces over which a stream flowed, cascading down from one level to another. The point of this portrayal was that water always flows downhill - it can't reverse itself. Similarly, the defining characteristic of the waterfall model is the irreversible forward progress from phase to phase.

One thing that you may notice is that some of the names of the phases in this process model are the same as the names of some of our workflows. That's unfortunate, but we pretty much have to live with it. You need to rely on context to know whether "Design" is referring to a workflow or to a phase of the development process. That should not really be all that difficult. It should be obvious in most cases whether what is being discussed is a low-level daily activity or a major phrase of development that may span several months.

1.1 Requirements analysis

Requirements analysis

This phase establishes what the customer requires from a software system. The primary product of this phase is a requirements document.

What is a requirement?

May range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification

- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation
  - May be the basis for the contract itself - therefore must be defined in detail
  - Both these statements may be called requirements

. . . . . .

Sample Reqts Specification

1.1 The user should be provided with facilities to define the type of external files
1.2 Each external file type may have an associated tool which may be applied to the file.
1.3 Each external file type may be represented as a specification from the user's display.
1.4 Facilities should be provided for the icon representing an external file type to be defined by the user.
1.5 When the user selects an icon representing an external file, the effect of that selection is to apply the tool associated with the type of the external file to the file represented by the selected icon.

. . . . . .

The requirements document

- The requirements document is the official statement of what is required of the system developers
  - Specify external system behaviour
  - Specify implementation constraints
  - Serve as reference tool for maintenance
- It is not a design document. As far as possible, it should indicate WHAT the system should do rather than HOW it should do it

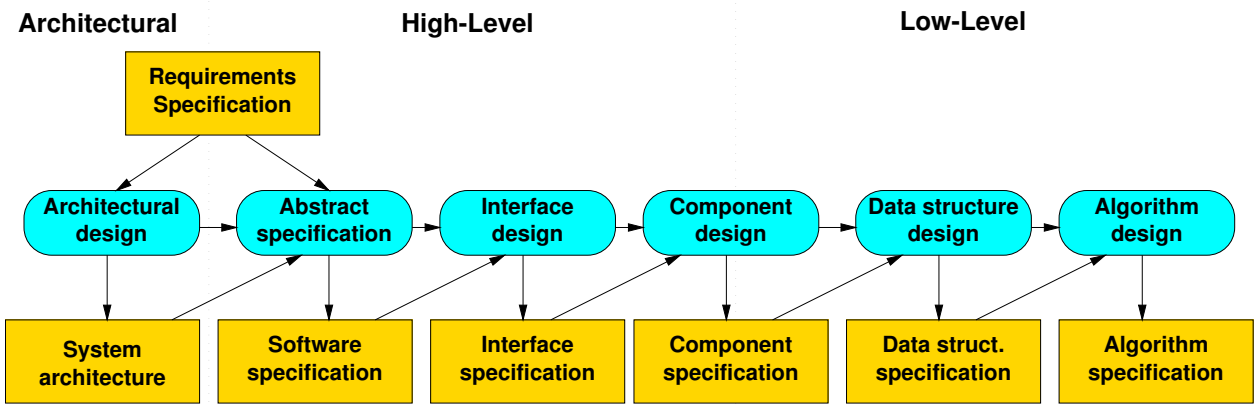
. . . . . .

1.2 Design

Design

This phase seeks to derive a solution which satisfies the software requirements. The primary product of this phase is design documentation.

Phases of design



Design can be broken down into three main phases

- *architectural design*, the collection of decisions that need to be common to all components.

  Examples of architectural design decisions would be

  - Will the system run on a single machine or be distributed over multiple CPUs?
  - Will outputs be stored in a file or in a database?

  - How will run-time errors be handled?

- *high-level* design, dividing the system into components, and
- *low-level* design (choosing the data structures and algorithms for a single component).

. . . . . .

Depending upon how formal a team's the overall process is, there may be several different design activities and corresponding design documents spread out over these three phases.

1.3 Implementation and unit testing

Implementation and unit testing

Writing the code – probably the most familiar activity.

The primary product of this phase is code – both application and test code.

. . . . . .

1.4 Verification and Validation

Verification and Validation

Verification & Validation: assuring that a software system meets the users' needs. The primary product of this phase is a test report.

The principle objectives are:

- The discovery of defects in a system
- The assessment of whether or not the system is usable in an operational situation.

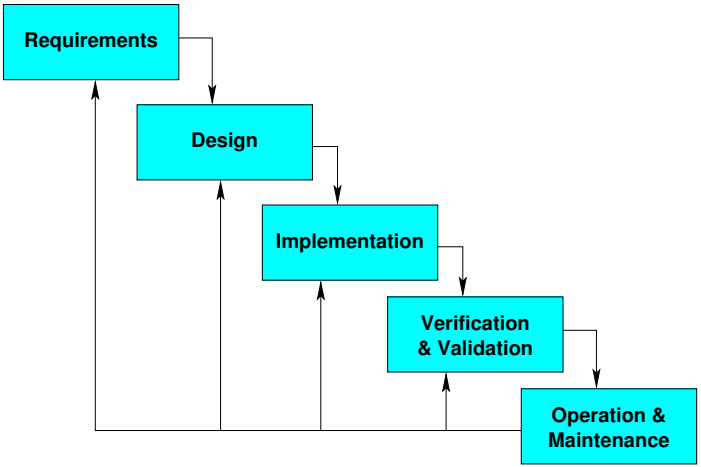
. . . . . .

V & V

Although the waterfall model shows this as a separate phase near the end, we know that some forms of V&V occur much earlier.

• Requirements are validated in consultation with the customers.

• Unit testing occurs during Implementation, etc.
So this phase of the waterfall model really describes system and acceptance testing.



1.5 Operation and maintenance

Operation and maintenance

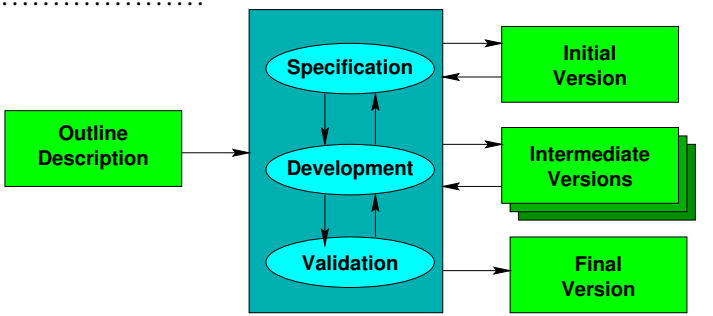
As requirements evolve and bug reports come in from the field, someone must

- prioritize changes
- make the changes
- validate the changes
  - new test cases
- validate that change does not break previously working code
  - regression testing

2 Iterative/Incremental Development

Iterative/Incremental Development

As a counter-reaction to what many believe to be an overly rigid waterfall model, there are a variety of incremental approaches that emphasize quick cycles of development, usually with earlier and more user-oriented validation.



Specification (analysis) and development are interleaved.

There is a greater emphasis on producing intermediate versions, each adding a small amount of additional functionality. Some of these are *releases*, either external (released outside the team) or internal (seen only by the team), which may have been planned earlier.

The evolutionary approach is conducive to exploration of alternatives. Some projects employ *throw-away prototyping*, versions whose code is only used to demonstrate and evaluate possibilities. This can lead to insight into poorly understood requirements.

Some waterfall projects may employ evolutionary schemes for parts of large systems (e.g., the user interface).

Risks of evolutionary schemes include poor process visibility (e.g., are we on schedule?), continual small drifts from the key architecture leading to poorly structured systems.

Incremental processes are periodically "rediscovered". For example, Extreme Programming is an evolutionary approach with heavy emphasis on involvement of the end-users in planning and on continuous validation.

Iterative Development

Advantages:

- conducive to exploration of alternatives

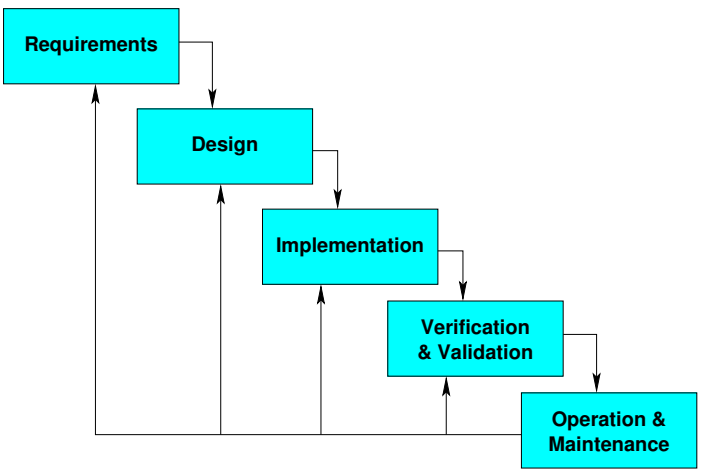
  - e.g., throw-away prototyping
  - can lead to insight into poorly understood requirements (e.g., GUIs)

Disadvantages:

- poor process visibility (e.g., are we on schedule?),
- continual small drifts from the key architecture leading to poorly structured systems.

. . . . . .

3 General OO Analysis & Design



If we were to replace the word "Requirements" in the traditional waterfall model with "Analysis", then you might be tempted to view this as an embodiment of our ADIV workflow cycle. That's not entirely accidental. In fact, many variations of the waterfall diagram actually refer to the first phase as "requirements analysis", because analysis, in the sense that we have been describing it (determining *what* the system needs to do), is the primary activity underlying the construction of a requirements document.

Now, the ADIV cycle is a low-level workflow model, but this suggests that something rather similar can be used as a strategic project plan by making the "topic of interest" the entire system. The idea of doing OOAD is not at all inconsistent with the traditional waterfall.

However, OOAD brings its own nuance to how these activities get carried out.

General OO Analysis & Design

OOAD is largely viewed as a model-building activity. The primary models that we find are

- domain model
- analysis model
- design model

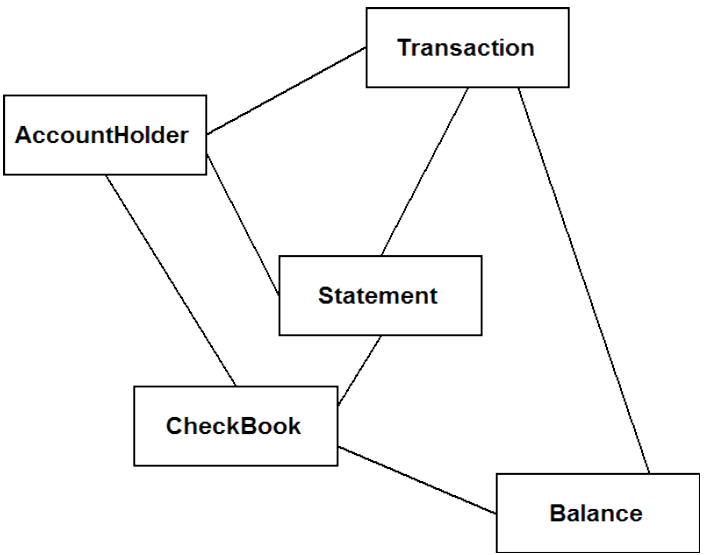
and OOAD emphasizes a smooth evolution from one to the other.

. . . . . .

3.1 Domain Models

Domain Models

A *domain model* is a model of the application domain as it currently exists, *before* we began our new development project. The point of the domain model is to be sure the development team understands the world that the system will work in.



- The domain model describes the world in terms of objects interacting with one another via messages.
  - We'll see techniques for capturing this kind of idea shortly.
- Not every project needs a domain model (e.g., If the team has done several projects already in this application domain)

. . . . . .

(Fortunately, most companies work in a small number of application domains. If they hired you last month to develop software for analyzing seismic data for petroleum engineering, they are unlikely to ask you to develop a compiler or a word processor next month.)

In that case, the team may already have a good understanding of the domain.

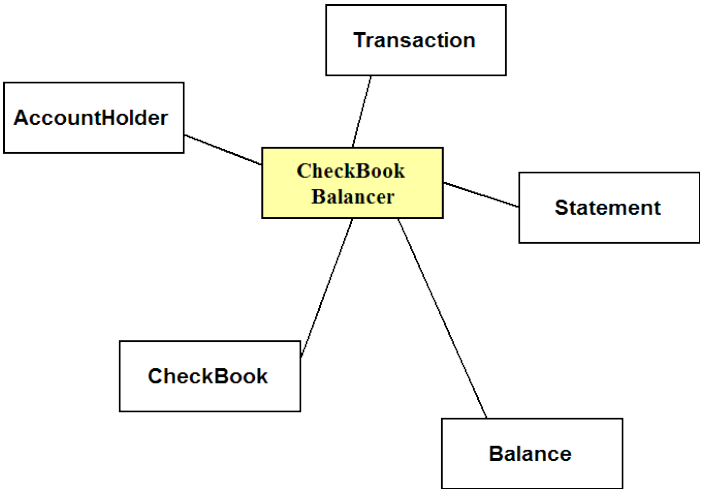
Even if a document describing the domain model is desired, domain models tend to be highly reusable since the world around our software systems usually changes fairly slowly.

3.2 Analysis Models

Analysis Models

An *analysis model* is a model of how the world will interact with the software system that we envision. As such, it is our statement of just *what* the system will do when it is working.

There is a real temptation to simply assume that the automated system will simply squat in the middle of the world, interacting with all the real world objects, sort of like this:
Or if you prefer,...

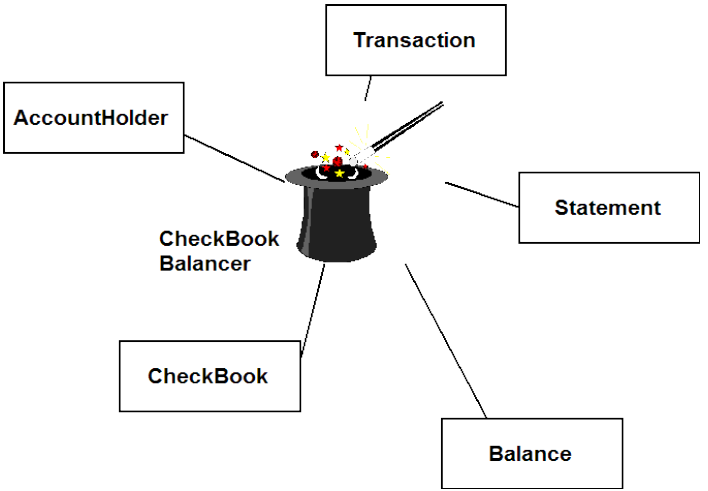


.....

It's Magic!

Poof! We have an analysis model!
Not wrong, per se, but it's certainly not helpful.
Such an approach is fundamentally at odds with the OO philosophy

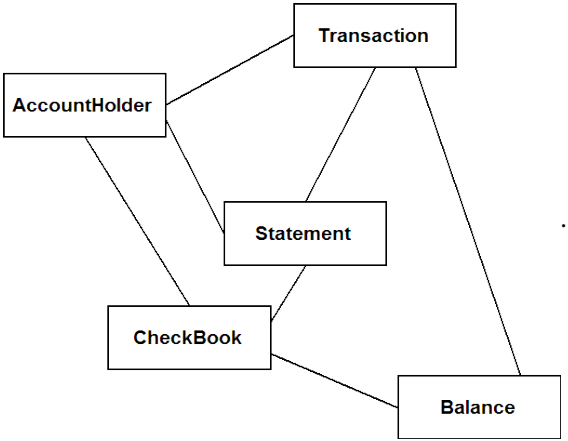
- We should look to the real world to suggest how to decompose our system.
- In this case, we have not decomposed it at all – just wrapped it up into a single black box.
  - All the hard decisions still need to be made.



In essence we have not done any analysis at all here. This "model" isn't *wrong*, per se, but it's certainly not helpful. We've basically thrown away everything we've learned in the domain model about how objects really interact. We're treating the new program as a simple box, with no knowledge of its internal structure, Essentially, we've just deferred all the hard questions to the upcoming design.

Evolving the Analysis Model

What we really hope for is an *evolution* from our domain model to our analysis model. The OO philosophy tells us that the classes and interactions of our domain model...



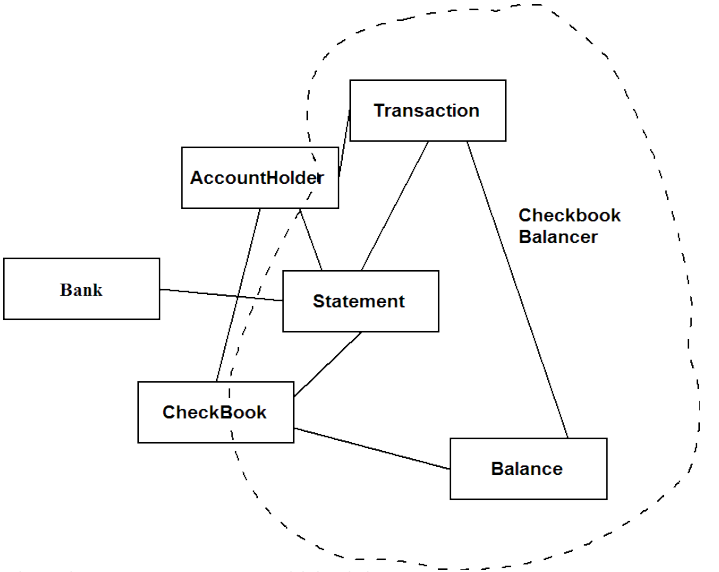
. . . should carry over into our analysis.

In essence, we hope to retain these classes, add more detail to our understanding of them, and to establish a boundary that tells us which of these classes and behaviors will be automated, which will remain entirely unautomated, and which will have some portion automated while other parts remain external.

The Boundary

...establish a boundary that tells us which of these classes and behaviors will

- be automated
- remain external
- be a mixture of the two



The system, then, remains a collection of interacting objects rather than an unstructured black box.

. . . . . .

There's a definite overlap in the purpose of a requirements document and of an analysis model. Some will regard the analysis model as a kind of requirements specification. In some projects, though, a requirements document will still be required as something for customers or management to sign off on. But the analysis model is the basis from which the eventual requirements document is derived.

3.3 Design Models

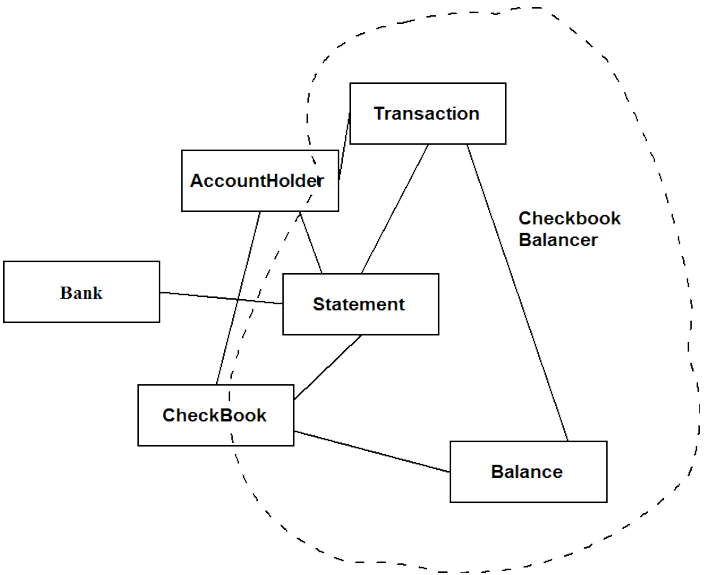
Design Models

Design is concerned with *how* we can get the system to do the things the analysis model says it should do.
Designs are expressed via a *design model*,
which, like the earlier models, also is based upon the idea of interacting objects. In fact, if we believe in the OO philosophy,

we would expect the objects and messages in the design model to bear a close resemblance to those from the analysis model. Hence we typically do not start building the design model from scratch so much as we *evolve* the analysis model into the design by adding detail, resolving conflicts, etc.

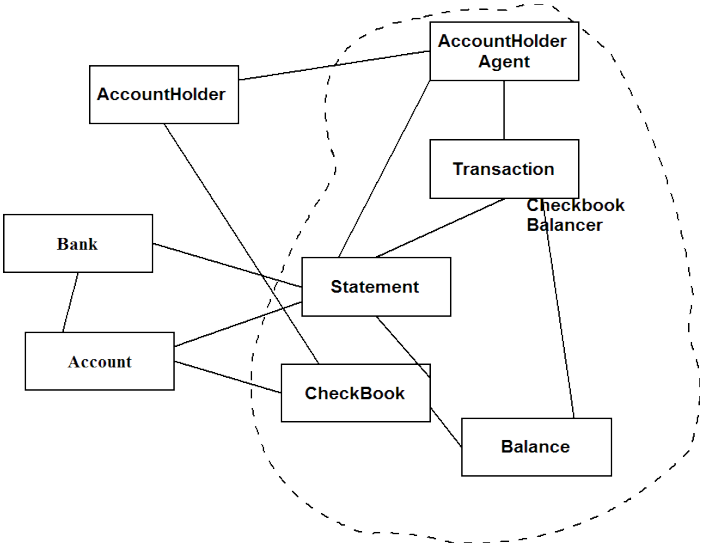
Design Model: Agents

A common step in OOD is to separate the classes that, in the analysis model, are partially implemented and partially external, into an purely external class representing the original, real-world objects, and an *agent* class that implements the automatable functions that will now be performed by the new system.



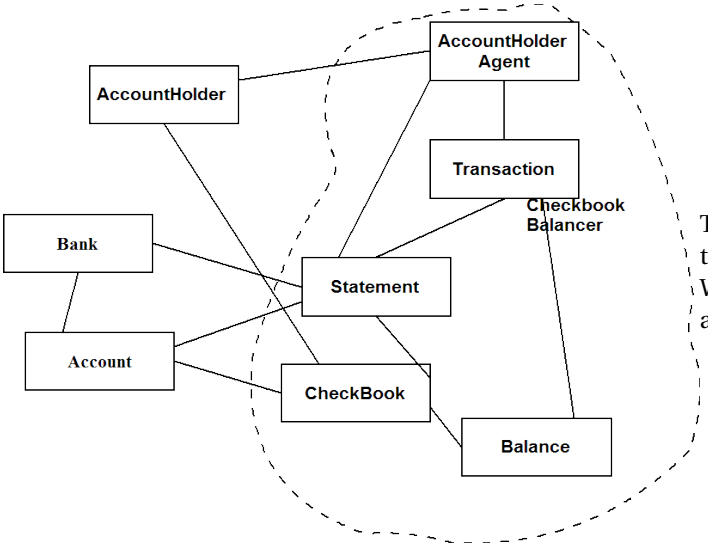
Design Model: Agents (2)

Here we have created an agent for AccountHolders



. . . . . .

Design Model: Interfaces



Then we can look closely at the lines (interactions) that cross the border from the outside world into the automated system. What do we call an interaction between an external entity and an internal automated entity?

Design Model: Interfaces

That's the definition of an "*interface*", and so

- each of those crossing points is a portion of the user interface (or an interface with other automated systems).

MVC

A common organizing principle for such interfaces is to distinguish between

- *viewers* that portray the contents of automated classes and
- *controllers* that accept external signals (e.g., mouse clicks, keyboard data) and interpret those signals as instructions to automated classes and to viewers.

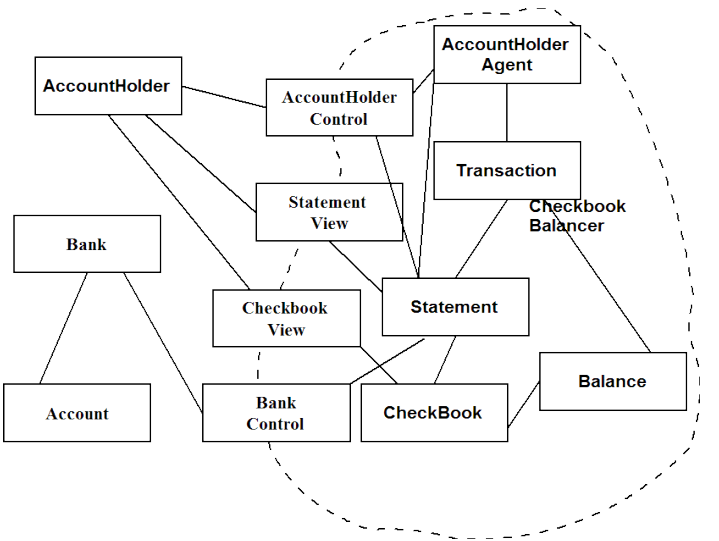
- Such viewers and controllers should be implemented as separate classes from the *model* classes that were derived from the analysis model.

.....

MVC Example

We then wind up with classes specifically devoted to managing the interface between the internal and external worlds.

- This is the Model-View-Controller pattern:



.....

Model-View-Controller

The MVC pattern establishes specific requirements so that GUI changes have minmal impact on the rest of the system.

- The model classes must not depend on (make use of the interface of) the view and controlelr classes.
- The view classes must not depend on the controller classes

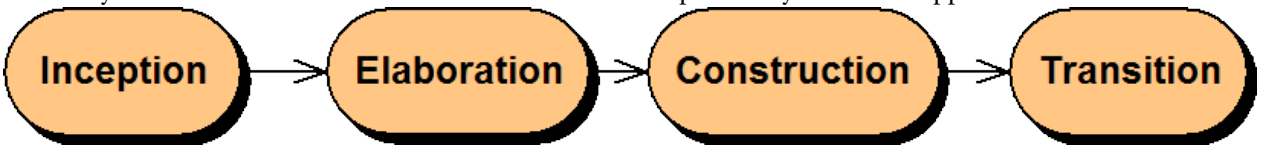
.....

4 The Unified Process Model

The Unified Process Model

For larger projects, we may want something a bit more formal than the general OOA&D process.

The *Unified Process* was developed by Jacobsen, Booch, and Rumbaugh, who were already some of the biggest names in OOA&D before they decided to collaborate on a unified version of their previously distinctive approaches.



Unified Model Phases

- Inception: initial concept
- Elaboration: exploring requirements
- Construction: building the software
- Transition: final packaging

4.1 Inception

Inception

- Pitching the project concept
- Usually informal, low details.
  - "Perhaps we should build a . . . "

4.2 Elaboration

Elaboration

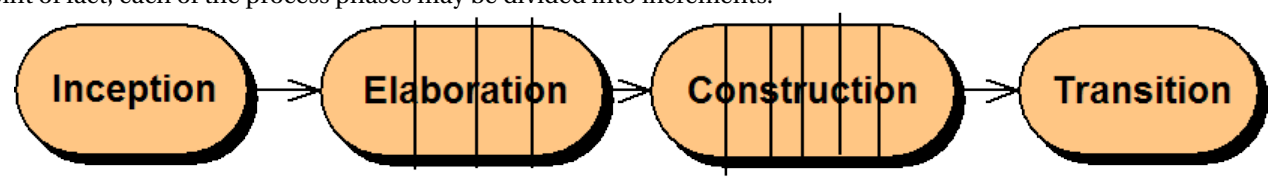
- Adding detail to our understanding of what the system should do.
- Produces
  - Domain model
  - Analysis model
  - Requirements document
  - Release plan

Among these products, only the release plan is new.

. . . . . .

Elaboration: Releases

In point of fact, each of the process phases may be divided into increments:



Such increments are most common in the elaboration and construction phases.

Each increment produces a *release* - some kind of product whose existence and/or acceptance by management shows that we are ready to move on.

A *release* represents the implementation of some part of the required functionality.

. . . . . .

Elaboration: Release Plans

The *release plan* records decisions about

- How many releases there will be
- What functionality will be added with each release
- When the releases will be made
- Which releases are internal (i.e., only the development team sees them) and which are external

. . . . . .

4.3 Construction

Construction

Perhaps the most familiar of the phases. This merges the waterfall activities of Design and Implementation.

- Each construction increment adds new functionality
- Documentation is constructed in addition to source code
- Each increment must be separately tested
- Each increment must be integrated and tested

. . . . . .

4.4 Transition

Transition

Activities that can't be done incrementally during construction, e.g.,

- performance tuning
- user training

. . . . . .

5 The Unified Process and Workflows

The Unified Process and Workflows

It should be noted again that the software development process model provides overall strategic guidance. The day-to-day "tactical" activity is still described by our four workflows.