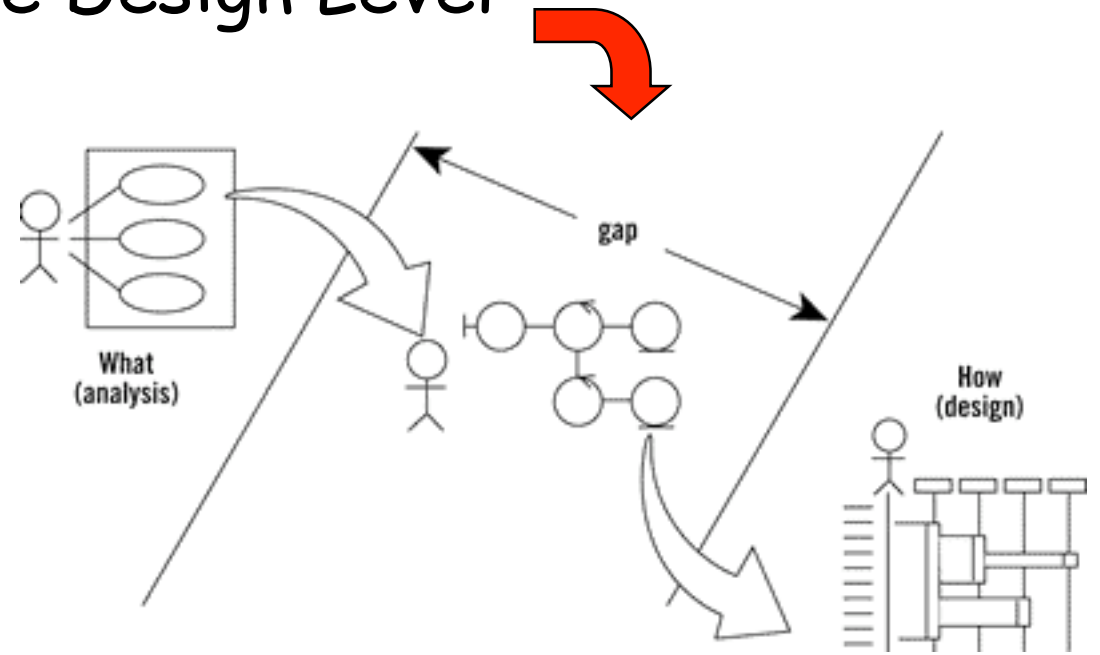
Jacobson's Analysis Model

(aka "Robustness Analysis")

What is Robustness Analysis

- An intermediate level of design, between Use Cases and Domain Classes, and the Software Design Level

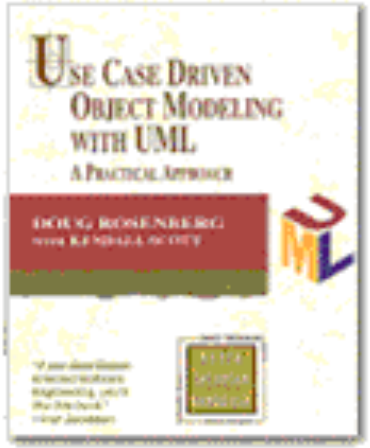


Who invented it?

- Ivar Jacobson (one of the "3 Amigos") in his Objectory Process (a forerunner of the Unified Process)
- But may have been derived from "Model-View-Controller" pattern, in Smalltalk lore

Is it part of UML?

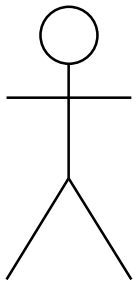
- No.
- It is not always used.
- However, there are proponents of it:
  - Doug Rosenberg, Iconix, author of *Use Case Driven Object Modeling with UML*
  - Recommends doing robustness analysis *before* sequence diagrams



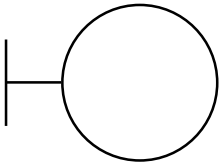
What is the Purpose?

- Provides a preliminary model for discussion.
- May lead to discovery of additional class needs.
- Clarifies collaborations based on "need to know"
- Provides a completeness or sanity check on use cases, before doing a full design.

Robustness Diagram Stereotypes (Icons)



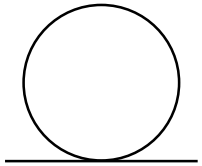
Actor



Interface Object

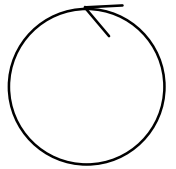
Object at the System Interface

also called "Boundary Objects"



Entity Object

Object representing stored data



Control Object

Object representing transfer of information

Jacobson: Interface Objects

- The task of an interface object is
  - to translate the actor's input into events in the system, and
  - to translate events in which the actor is interested into something that can be presented to the actor.
- Each actor should have its own interface, and some may need multiple.

Jacobson: Entity Objects

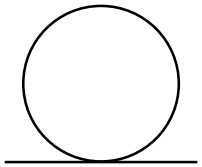
- Entity objects model information that is kept long term, e.g. between use cases.

Jacobson: Control Objects

- Control objects typically act as "glue", uniting the other objects so that they represent one use case.
- They are typically the most ephemeral, lasting only as long as the performance of one use case lasts.

Related Idea: "Model-View-Controller"

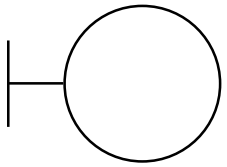
(originally from Smalltalk system developers)



Entity Object

Object representing stored data

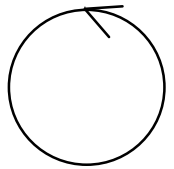
"Model"



Interface Object

Object at the System Interface

"View"



Control Object

Object representing transfer of information

"Controller"

The Idea of Model-View-Controller

- Guiding Principle: **Separation of Concerns**
- The **model** captures the core data characteristics, but does not attempt to capture all ways in which the model can be used.
- There may be multiple **views** of a given model.
- **Controllers** provide the ways to update and extract information from the model.

Example of Robustness Diagram: A Managed Version Control System

- On some projects, the baseline source code is not purely under control of programmers.
- Instead it is desired to have **managerial control** over what code is or is not in the system.

Example: Managed Version Control (entails several use cases)

- In order to manage changes in a disciplined way, changes are made through

  "Change Packages"

  each of which consists of :
  - new files to be **added**,
  - files to **replace** existing files, and
  - explicit directions for **removal** of files.

Example: Managed Version Control:

- The **Project Manager** authorizes the development of a **Change Package**, which starts in the **development state**.
- A **Developer** develops code for the Change Package.
- When the developer has coded and tested the package, he/she "promotes" the package to the **development completed** state.
- It is then up to the **Build Manager** to apply the CP to the current **Baseline Configuration**, forming a **Build**.

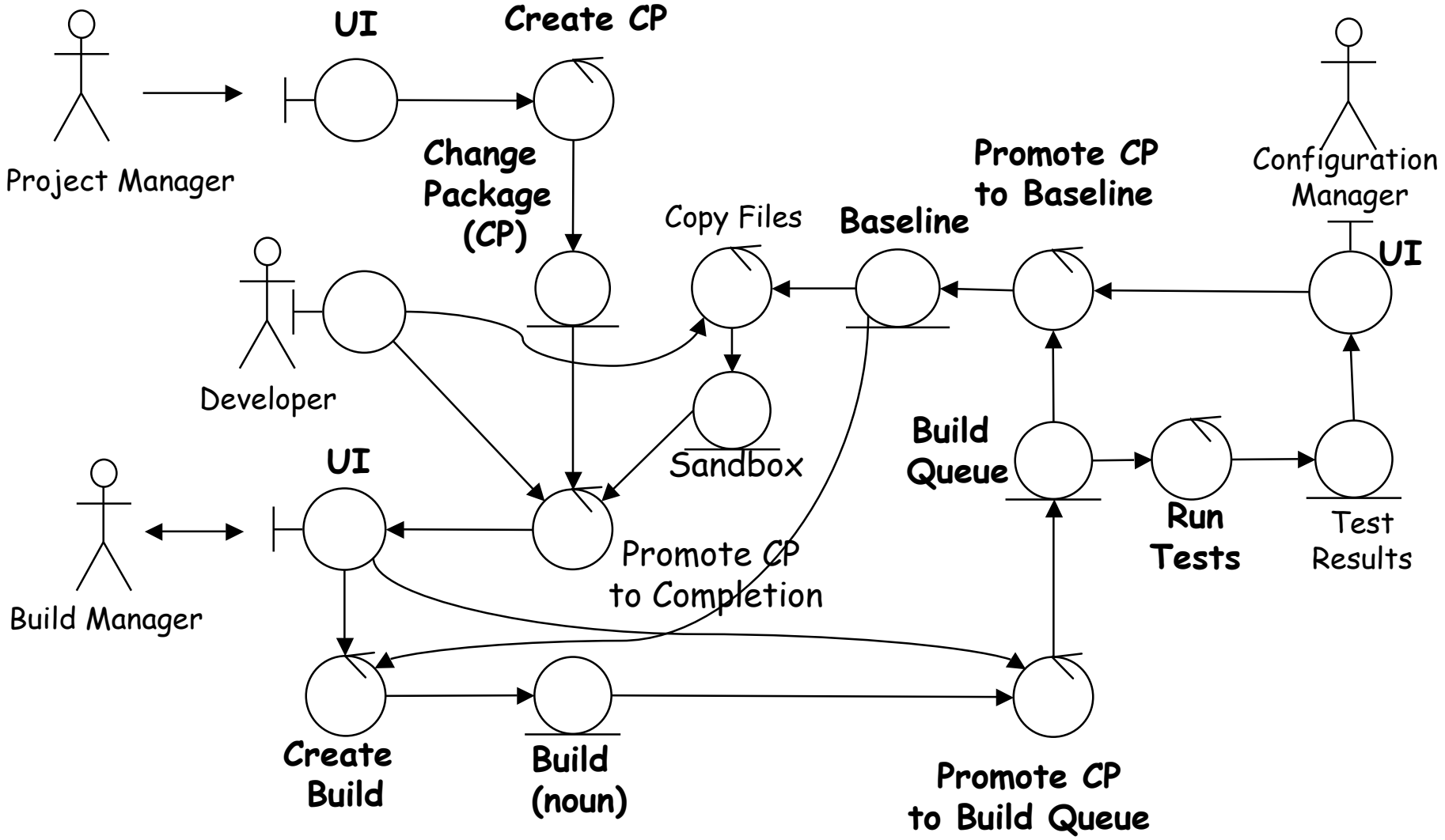
Example: Managed Version Control:

- The Build Manager subjects the new Build to further tests.
- If the tests are passed, the Build Manager may promote the CP to the Baseline.
- (If tests are not passed, the CP is automatically **demoted** back to the Development state. The current diagram will not show this exception.)

Promotion Scenario

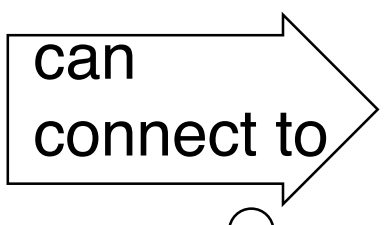
- Developer promotes CP to completion.
- Build Manager notified.
- Build Manager creates Build from Baseline and CP.
- Build Manager adds Build to Build Queue.
- Tests are run automatically.
- If tests are passed, Build Manager is notified.
- Build Manager adds CP to Baseline.

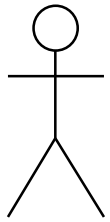
Robustness Diagram for Package Use Case

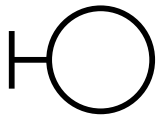


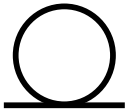
Robustness Diagram Connectivity Rules

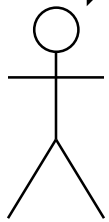
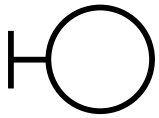
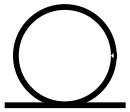
(Rosenberg)









	no	YES	no	no
	YES	no	YES	no
	no	YES	YES	YES
	no	no	Yes	no

Purpose of these Rules

- It is better if entity classes that can be understood in isolation did not have to "know about" each other.
- Instead, **control classes** can be interposed that know about both entity classes.
- This philosophy is again consistent with the Model-View-Controller pattern.
- It is also related to the "Law of Demeter" (more on this later).

Paraphrasing the Principle

- If two entity classes interact in this application or use case, but
- generally don't need to know about each other, then
- it is better to connect them with a controller class.

Component Architecture

- The same principle seems to underlie what is called "Component-based Software Architecture":

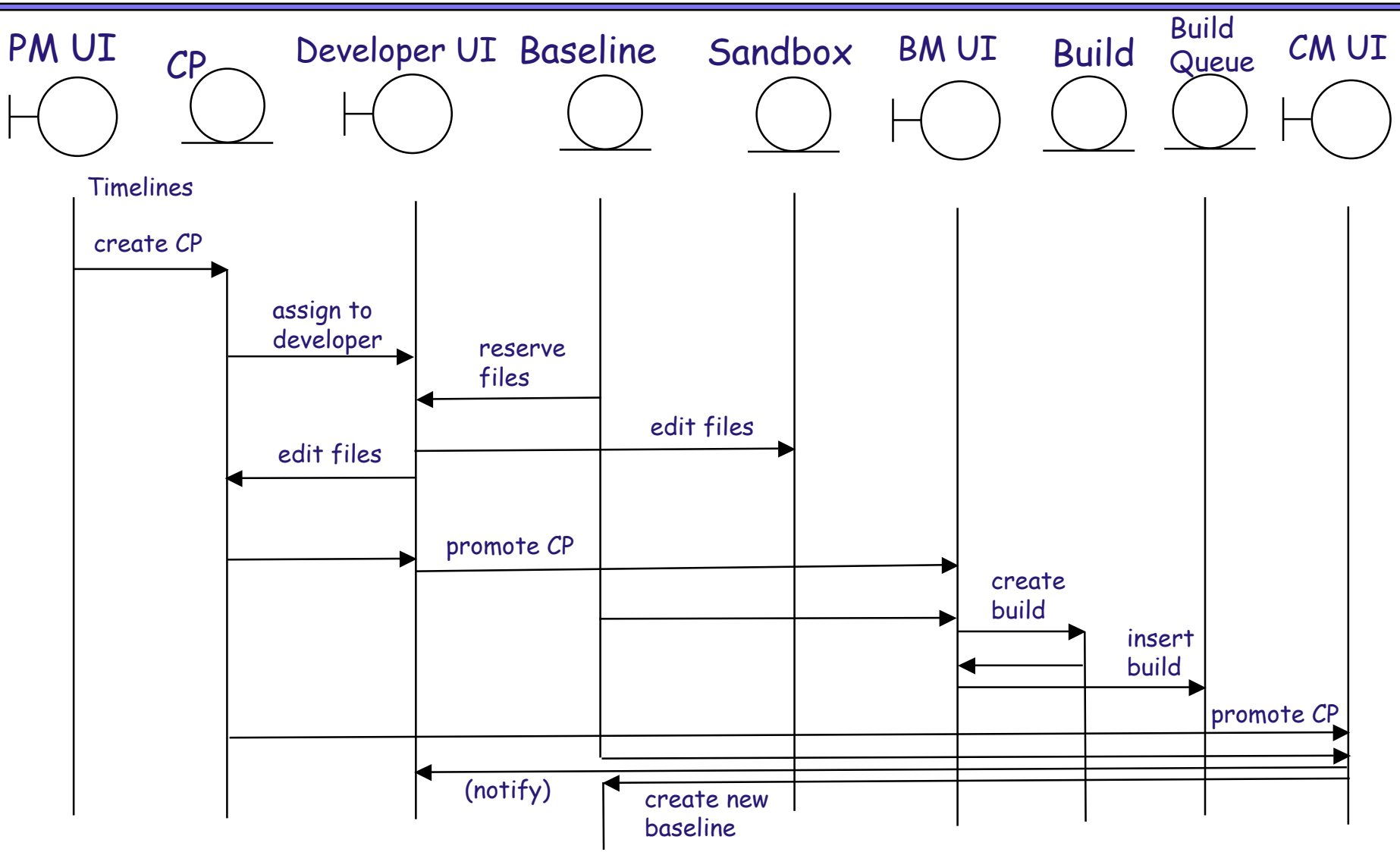
  component ~ interface or entity

  connector ~ controller

Per Doug Rosenberg

- Domain analysis is not complete until you can
  - Construct a robustness diagram that includes the domain classes.
  - **Trace the use cases** out on the robustness diagram.
  - Constructing a **sequence diagram** at this point might also be helpful.
- (However, Rosenberg does not advocate keeping robustness diagrams up-to-date following initial analysis. I'm not sure that I agree.)

Sequence Diagram



Rosenberg: Benefits of Robustness Analysis

- Closes semantic gap between use cases and design.
- Forces you to write use cases in a consistent style, and in the correct voice.
- Provides sanity, syntax, and completeness checks for use cases, before launching formal design.
- Provides guidelines for partitioning system (MVC, or client-server).
- Provides traceability between use cases and operation of the system.

More Info

- Ivar Jacobson, et al. Object-Oriented Software Engineering, A Use Case Driven Approach, revised printing, ACM Press, Addison-Wesley, 1997.
- Doug Rosenberg, with Kendall Scott, Use Case Driven Object Modeling with UML, Addison-Wesley, 1999.
- http://www.ddj.com/article/printableArticle.jhtml?articleID=184414712&dept_url=/architect/ (by Rosenberg)
- www.bredemeyer.com/whatis.htm (on software architecture)