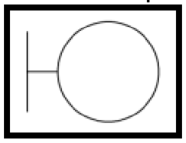
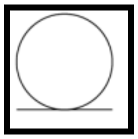
Robustness Analysis:

- **Law of Demeter:**
- The **Law of Demeter** states that a module should not have the knowledge on the inner details of the objects it manipulates. In other words, a software component or an object should not have knowledge of the internal working of other objects or components.
- The Law of Demeter reduces dependencies and helps build components that are loosely coupled for code reuse, easier maintenance, and testability.
- I.e.
  A method, M, of an object, O, can only call methods of:
  - O itself
  - M's parameters
  - Any objects created by M
  - O's direct component objects

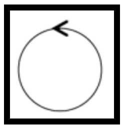
  M cannot call methods of an object returned by another method call.
- **Note:** The programmer's rule of thumb is to use 1 dot.
  E.g. Instead of Customer.PayPalAccount.CreditCard.Subtract(total), use Customer.getPayment(total).
- **Robustness Analysis:**
- **Robustness analysis** is a way of analyzing your use case model. Robustness analysis provides an approach to the structuring of problem situations in which uncertainty is high, and where decisions can or must be staged sequentially. The specific focus of robustness analysis is on how the distinction between decisions and plans can be exploited to maintain flexibility. These are classified into **boundary objects**, **entity objects**, and **control objects**.
- **Boundary Objects:**
- Used by actors when communicating with the system.
- Represents the interfaces between the actors and the system.
- The "view" part of MVC architecture.



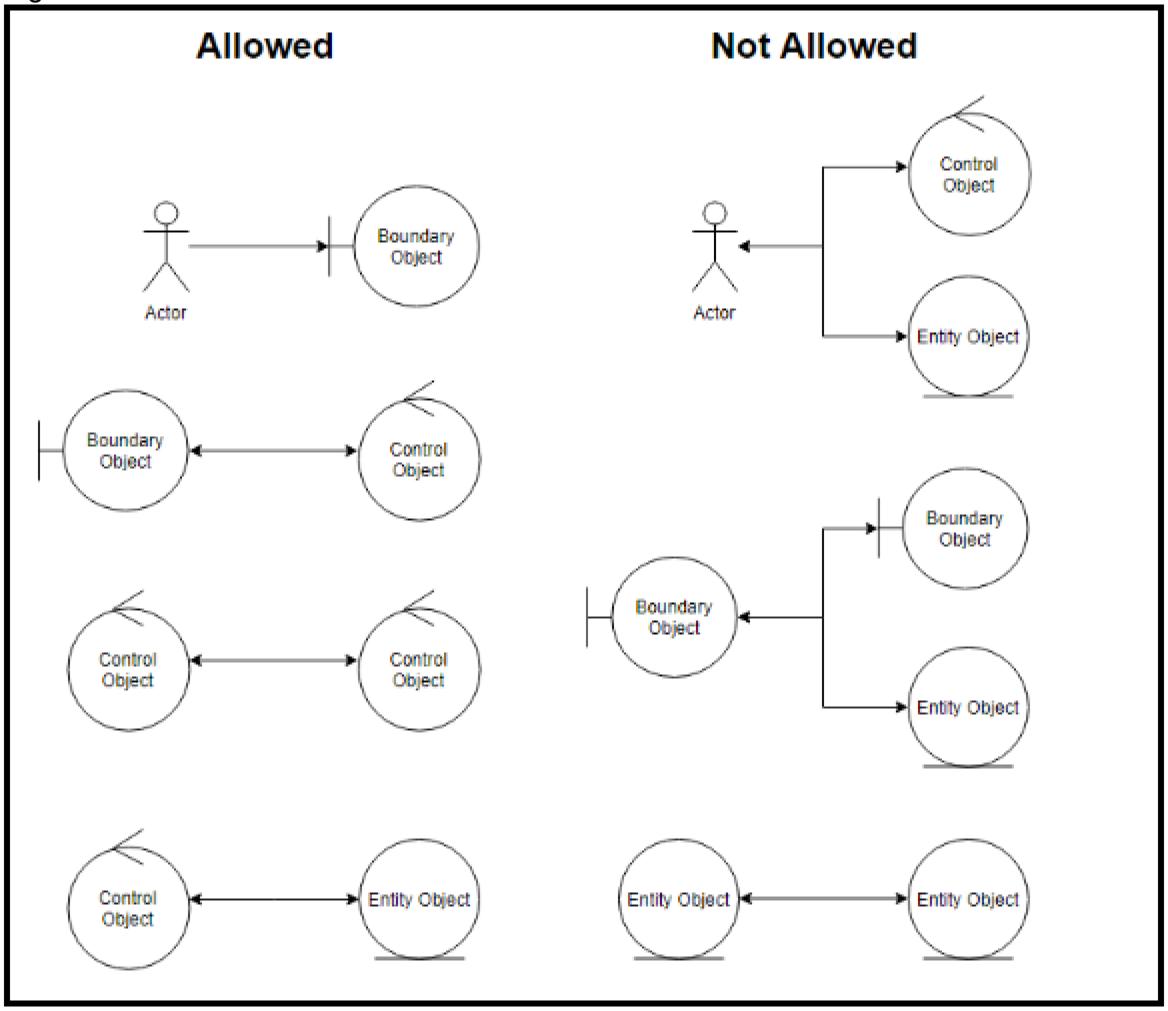
- **Entity Objects:**
- Usually objects from the domain model. They are objects representing stored data.
- Manages the information the system needs to provide the required functionality.
- The "model" part of the MVC architecture.



- **Control Objects:**
- The "glue" between boundary objects & entity objects. It captures business rules and policies and represents the use case logic and coordinates the other classes.
- The "controller" part of the MVC architecture.
- **Note:** It is often implemented as methods of other objects.



- **Robustness Diagram – 4 Connection Rules:**
- Boundary objects and entity objects are nouns while controllers are verbs. Nouns can't talk to other nouns, but verbs can talk to either nouns or verbs. Here are the four basic connection rules which should always be mind:
    1. Actors can only talk to boundary objects.
    2. Boundary objects can only talk to controllers and actors.
    3. Entity objects can only talk to controllers.
    4. Controllers can talk to boundary objects, entity objects, and other controllers, but not to actors.
- E.g.

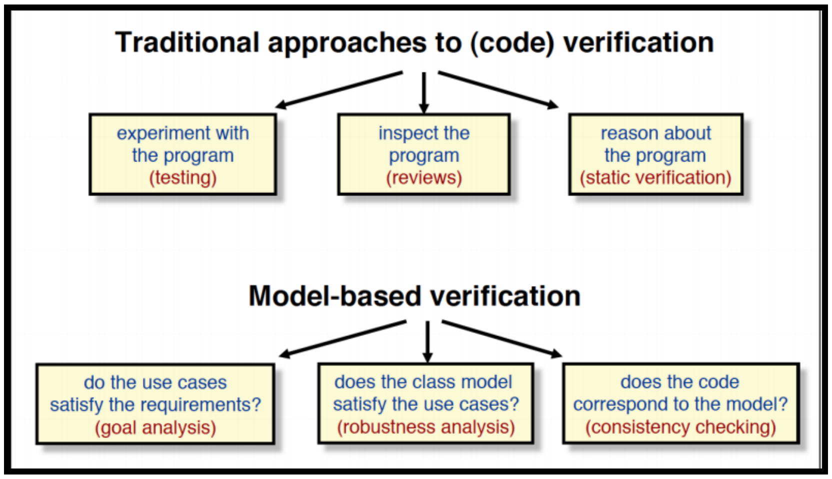


- **Why Use Robustness Analysis:**
    1. Bridges the gap between design and requirements.
    2. Sanity Check:
        - Tests the language in the use case description.
        - Nouns from the use case get mapped onto objects.
        - Verbs from the use case get mapped onto actions.
    3. Completeness Check:
        - Discover the objects you need to implement the use case.
        - Identify alternative courses of action.
    4. Object Identification:
        - Decide which methods belong to which objects.

- **Benefits of Robustness Analysis:**
    1. Forces a consistent style for use cases.
    2. Forces a correct 'voice' for use cases.
    3. Sanity and completeness check for use cases.
    4. Creates syntax rules for use case descriptions.
    5. Quicker and easier to read than sequence diagrams.
    6. Encourages use of Model-View-Controller (MVC) pattern.
    7. Helps build layered architectures e.g presentation layer, domain layer, repository layer.
    8. Checks for reusability across use cases before doing detailed design.
    9. Provides traceability between user's view and design view.
    10. Plugs semantic gap between requirements and design.
- **Constructing a Robustness Diagram:**
    1. Add a boundary element for each major UI element. Not at the level of individual widgets though.
    2. Add controllers:
        - One to manage each use case.
        - One for each business rule.
        - Another for each activity that involves coordination between several other elements.
    3. Add an entity for each business concept.

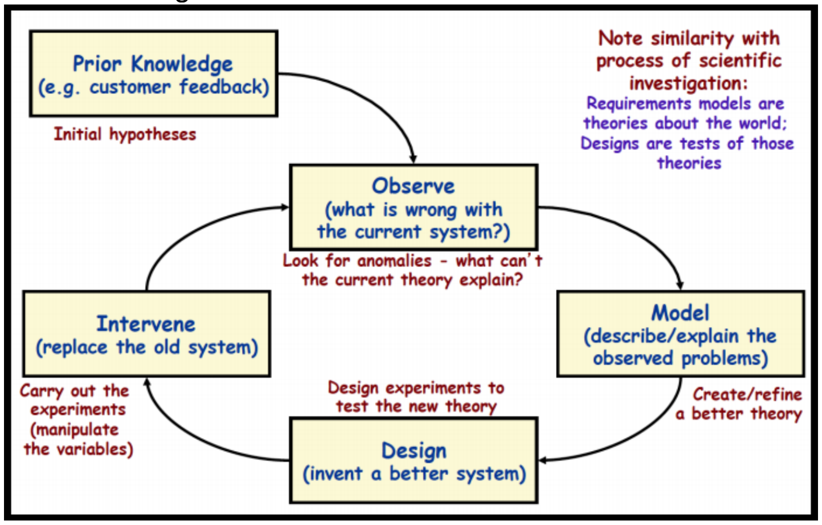
__**Verification and Validation (V&V):**__

- **Verification:**
- **Verification** is testing that your product meets the specifications/requirements you have written.
  I.e. Are we building the system right?
  It is ensuring that the software to be built is actually what the user wants.
- I.e.

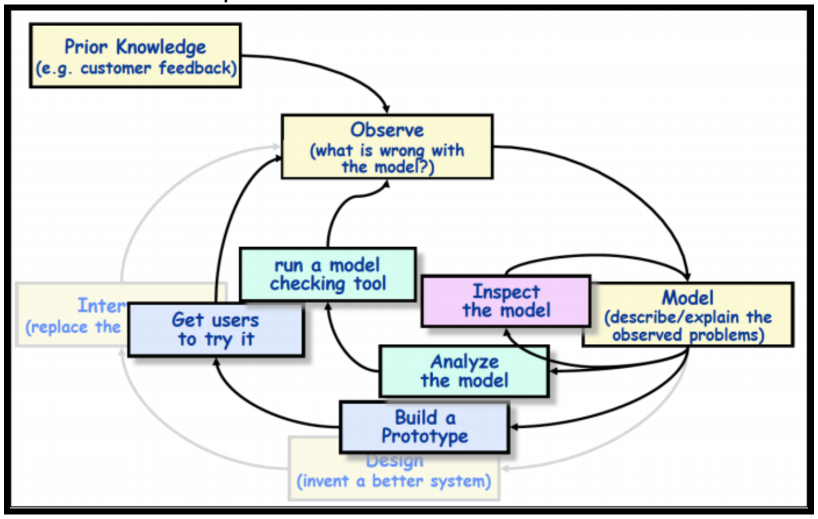

Traditional approaches to (code) verification
experiment with the program (testing)
inspect the program (reviews)
reason about the program (static verification)
Model-based verification
do the use cases satisfy the requirements? (goal analysis)
does the class model satisfy the use cases? (robustness analysis)
does the code correspond to the model? (consistency checking)

- **Validation:**
- **Validation** is testing how well you addressed the business needs that caused you to write those requirements. It is also sometimes called acceptance or business testing.
  I.e. Are we building the right system?
  It is ensuring that the software runs correctly.

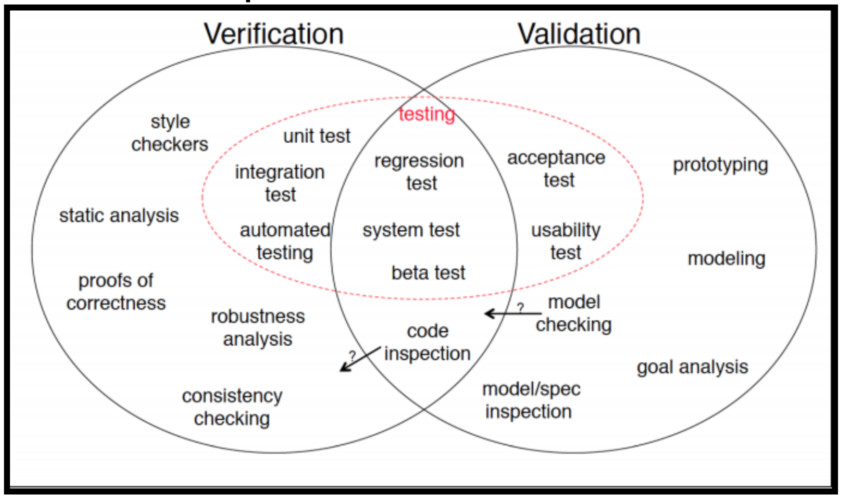
- Understanding validation:


Prior Knowledge (e.g. customer feedback)
Initial hypotheses
Note similarity with process of scientific investigation:
Requirements models are theories about the world; Designs are tests of those theories
Observe (what is wrong with the current system?)
Look for anomalies - what can't the current theory explain?
Intervene (replace the old system)
Model (describe/explain the observed problems)
Carry out the experiments (manipulate the variables)
Design experiments to test the new theory
Create/refine a better theory
Design (invent a better system)

- Validation techniques:


Prior Knowledge (e.g. customer feedback)
Observe (what is wrong with the model?)
run a model checking tool
Inspect the model
Model (describe/explain the observed problems)
Inter (replace the
Get users to try it
Analyze the model
Build a Prototype
Design (invent a better system)

- **Choice of Techniques:**



- **Roles for Independent V&V:**
- V&V is usually performed by an independent contractor.
- Independent verification and validation involves V&V done by a third party organization not involved in the development of the product. The main check performed is whether user requirements are met alongside ensuring that the product is structurally sound and built to the required specifications. Independent V&V fulfills the need for an independent technical opinion.
- V&V costs between 5% and 15% of development costs, but a study by NASA showed that its return on investment is fivefold. Bugs/errors are found earlier, making them easier to fix. The specifications are clearer. Developers are more likely to use better practices.
- 3 types of independence:
    1. Managerial Independence:
        - Separate responsibility from that of creating the software.
        - Can decide when and where to focus the V&V effort.
    2. Financial Independence:
        - Costed and funded separately.
        - No risk of diverting funds/resources when the going gets tough.
    3. Technical Independence:
        - The code is looked at by an outside party. There is little or no bias.
        - The tools and techniques used can be different.
- **Prototyping:**
- Presentation Prototypes:
    - Used for proof of concept and explaining design features.
- Exploratory Prototypes:
    - Used to determine problems, elicit needs, clarify goals, and compare design options.
    - It is informal and unstructured.
- Breadboards or Experimental Prototypes:
    - Used to explore technical feasibility or to test the suitability of a technology.
    - Typically there is no user/customer involvement.
- Evolutionary/Operational Prototypes:
    - Development is seen as a continuous process of adapting the system.
- **Usability Testing:**
- Real users try out the system or prototype and write down what problem(s) they observed. 3-5 users give the best return on investment.
- **Model Analysis:**
- Verification:
    - “Is the model well-formed?”
    - Are the parts of the model consistent with one another?
- Validation:
    - ‘What if’ questions:
        - Reasoning about the consequences of particular requirements.
        - Reasoning about the effect of possible changes.
        - Asking if the system will ever do the following tasks.
    - Formal challenges:
        - If the model is correct then the following property should hold.
    - Animation of the model on small examples
    - State exploration:
        - Using model checking to find traces that satisfy some property.

- **UML Consistency Check:**
- Use Case Diagrams:
  - Does each use case have a user?
  - Is each use case documented?
- Class Diagrams:
  - Does the class diagram capture all the classes mentioned?
  - Does every class have methods to set and get its attributes?
- Sequence Diagrams:
  - Is each class in the sequence diagram?
  - Can each message be sent?
    - Is there an association connecting sender and receiver classes on the sequence diagram?
    - Is there a method call in the sending class for each message sent?
    - Is there a method call in the receiving class for each message received?
- **Model Checkers:**
- Automatically check properties (expressed in Temporal Logic):
  - $\boxed{\Box p}$ means that p is true now and always (in the future).
  - $\boxed{\Diamond p}$ means that p is true eventually (in the future).
  - $\boxed{\Box(p \Rightarrow \Diamond q)}$ means that whenever p occurs, it's always (eventually) followed by a q.
- The model may be:
  - Of the program itself (each statement is a 'state').
  - An abstraction of the program.
  - A model of the specification.
  - A model of the requirements.
- A model checker searches all paths in the state space with lots of techniques for reducing the size of the search.
- Model checking does not guarantee correctness.
  It only tells you about the properties that you ask about and may not be able to search the entire state space.
  However, it is good at finding many safety, liveness and concurrency problems
- **Inspections:**
- Management Reviews:
  - Used to provide confidence that the design is sound.
  - The audience are the management and sponsors (customers).
  - Examples include preliminary design reviews and critical design reviews.
- Walkthroughs/Scientific Peer Reviews:
  - Used by development teams to improve the quality of the product.
  - The focus is on understanding design choices and finding defects.
- Fagan Inspections:
  - A process management tool. (Always formal)
  - Used to improve the quality of the development process.
  - Collect defect data to analyze the quality of the process.
  - Written output is important.
  - Major role in training new staff and transferring expertise.

- Inspections are very effective. Code inspections are better than testing for finding defects. For models and specifications, inspections ensure that the domain experts carefully reviewed them.
- Key ideas of inspections:
    - Preparation: Reviewers individually inspect the code/diagram first.
    - Collection meeting: Reviewers meet to merge their defect lists. They note each defect, but don't spend time trying to fix it. The meeting plays an important role as reviewers learn from one another when they compare their lists and additional defects can be uncovered. Defect profiles from each inspection are important for process improvement.
- How to structure the inspection:
    - Checklist:
        - Uses a checklist of questions/issues.
        - The review is structured by the issues on the list.
    - Walkthrough:
        - One person presents the product step-by-step.
        - The review is structured by the product.
    - Round Robin:
        - Each reviewer gets to raise an issue. (Goes in a circle)
        - The review is structured by the review team.
    - Speed Review:
        - Each reviewer gets 3 minutes to review a chunk and then passes it to the next reviewer.
        - Good for assessing comprehensibility.
- Benefits of inspection:
    - For applications programming:
        - More effective than testing.
        - Most reviewed programs run correctly the first time.
    - Data from large projects:
        - Error reduction by a factor of 5 (10 in some reported cases).
        - Improvement in productivity by 14% to 25%.
        - Percentage of errors found by inspection: 58% to 82%.
        - Cost reduction of 50%-80% for V&V even including cost of inspection.
    - Effects on staff competence:
        - Increased morale, reduced turnover.
        - Better estimation and scheduling (more knowledge about defect profiles).
        - Better management recognition of staff ability.