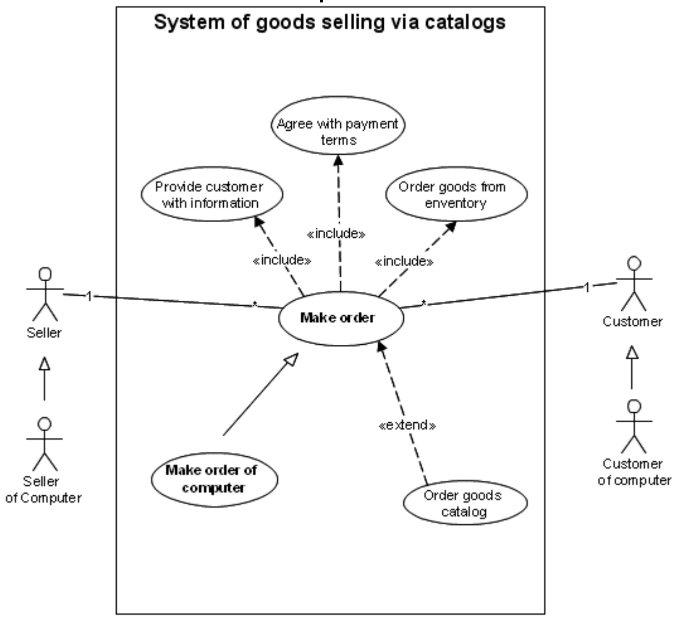
Design Methodologies

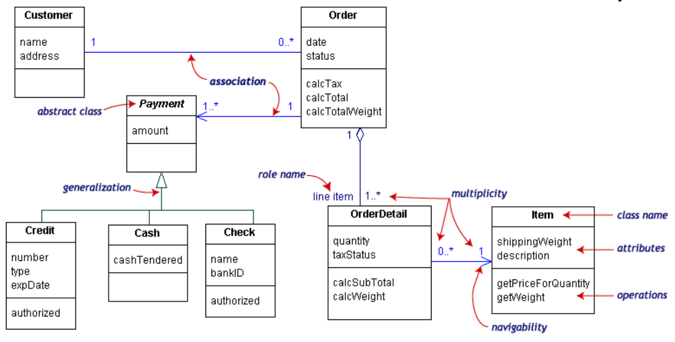
Waterfall	Requirements (user language) Specification (system language) Implementation / unit testing (checks units against spec) Integration / system testing (checks requirements met) Operations and maintenance
Spiral	Plan (requirements, feedback) Determine objectives / alternatives Evaluate alternative / risks (prototype) Develop / verify (code / test / integrate)
User-Centred	Design a shared conceptual model of the system with the user Anthropology (interview users etc) Collaborate to decide what to solve System mock ups / talk-through CRC (responsibility & collaborators)
JSP	Program structure in terms of data
XP	Pair programming, agile, get feedback from users ASAP Refactor the design when requirements change

UML

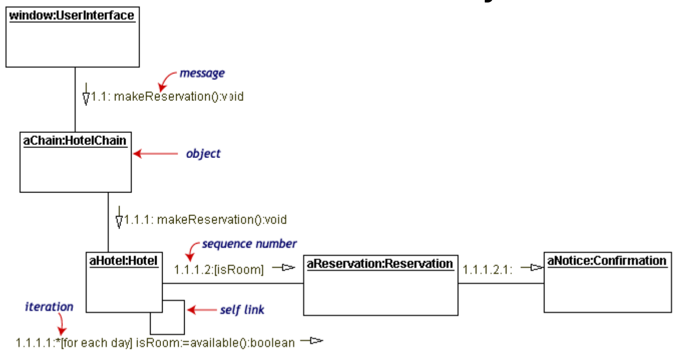
Use Case	Describe the human activity that the system has to support Focal point of discussion



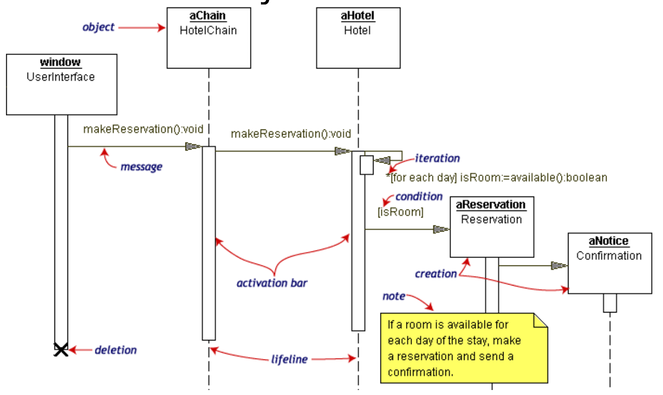
Class	Shows classes & relationships



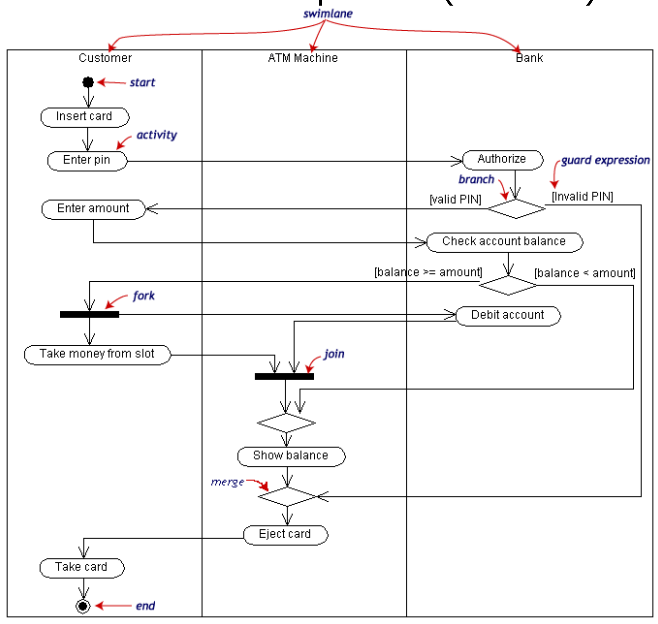
Collaboration	Interaction diagrams focusing on the roles of objects



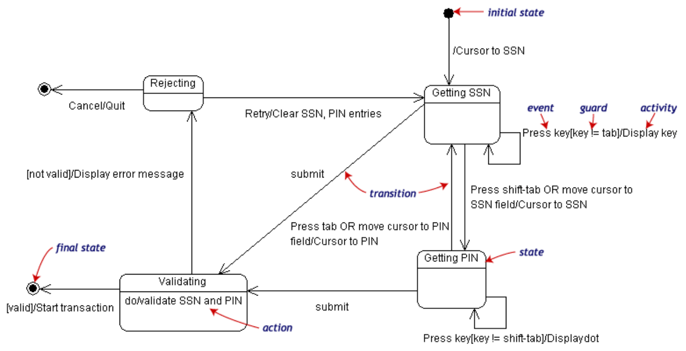
Sequence	Interaction diagrams focusing on the time at which messages are sent between objects



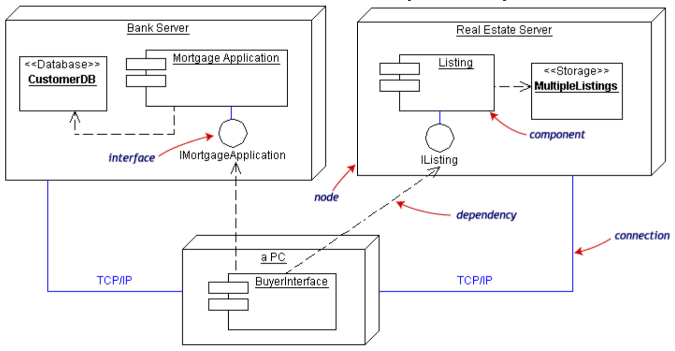
Activity	Shows how object activities are dependent (flowchart)



Statechart	Show object lifecycle and internal state transitions


initial state
/Cursor to SSN
Rejecting
Cancel/Quit
Retry/Clear SSN, PIN entries
Getting SSN
event
guard
activity
Press key[key != tab]/Display key
[not valid]/Display error message
submit
transition
Press shift-tab OR move cursor to SSN field/Cursor to SSN
Press tab OR move cursor to PIN field/Cursor to PIN
final state
Getting PIN
state
Validating
do/validate SSN and PIN
[valid]/Start transaction
submit
action
Press key[key != shift-tab]/Displaydot

Component	Dependencies between components (could include web pages, EXEs, DLLs etc.)
Deployment	Location of components that make a complete system



Object Design

Information Hiding	Expose minimum interface No implementation details
Loose Coupling	Minimize object joins
Cohesion	"1 method does 1 thing"
Abstract Types	Hide implementation Interface as specification
Modularization	Separate building / testing Helps code reuse Divide work between teams Change localisation
Method Classification	Mutator and accessor
Responsibility	Classes manipulate own data
Defensiveness	Performance/redundancy tradeoff w/ multiple check How to deal with failure? Exceptions can enforce this Attempt error avoidance

Correctness

Typing	Strong typing can catch errors at compile time Variable "taxonomy"
Formal Models	Define program element If precondition holds then execution means the postcondition holds Composition possible Force fine level analysis Static checking / verification Alternative (declarative) perspective Specialist (Greek letters!) LOD similar to that of code
Patterns	Reusable approaches
Testing	Regressions / unit tests